Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Klemen Možina

Razvoj odzivnih kompozitnih komponent v JSF

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Proučite tehnologijo JSF za izgradnjo dinamičnih spletnih aplikacij. Podrobno analizirajte koncept kompozitnih komponent, identificirajte ustrezne JSF oznake in načine razvoja ter izgradnje naprednih kompozitnih komponent. Proučite tehnologije na strani odjemalca za izgradnjo odzivnih spletnih aplikacij in analizirajte njihovo uporabo v povezavi z JSF. S pomočjo JSF in tehnologij na strani odjemalca izdelajte štiri nove kompozitne komponente, jih opišite in preizkusite.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Klemen Možina, z vpisno številko **63110300**, sem avtor diplomskega dela z naslovom:

Razvoj odzivnih komponzitnih komponent v JSF

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Matjaža Branka Juriča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Prestranku, dne 5. septembra 2014 Podpis avtorja:

Mentorju prof. dr. Matjažu Branku Juriču se najlepše zahvaljujem za njegovo podporo, dosegljivost in potrpežljivost kot tudi za motivacijo, ki mi jo je dal pri delu.

Hvala tudi staršem in bratu za vso podporo med študijem.

Kazalo

Seznam uporabljenih kratic

kratica	pomen
AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CDI	Context Dependency Injection
CSS	Cascading Style Sheets
DOM	Document Object Model
DRY	Don't Repeat Yourself
EJB	Enterprise JavaBeans
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JAVA EE	Java Enterprise Edition
JPA	Java Persistence API
JSF	JavaServer Faces
JSP	JavaServer Pages
JSON	JavaScript Object Notation
JSTL	JavaServer Pages Standard Tag Library
MVC	Model View Controller
RIA	Rich Internet Application
UI	User Interface
URL	Uniform Resource Locator
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language

Povzetek

Odzivnost spletnih aplikacij je v današnjem času zaradi porasta števila mobilnih naprav ena izmed glavnih tematik na področju spletnega razvoja. Na področju principov razvoja programske opreme pa se poudarja modularen razvoj po komponentah, ki so ponovno uporabljive. V diplomskem delu je podrobno opisan razvoj odzivnih kompozitnih komponent v tehnologiji JavaServer Faces (JSF), pri katerem razvijemo samostojne komponente, ki so ponovno uporabljive in jih lahko vključimo v vsako spletno aplikacijo JSF. Poleg tega pa so komponente odzivne, s čimer je spletnim razvijalcem prikrajšano to, velikokrat neprijetno, delo. Na začetku sta podrobno opisana delovanje tehnologije JSF in razvoj kompozitnih komponent, ki mu sledi opis uporabljenih tehnologij za dosego odzivnosti. Sledi izčrpen prikaz razvoja štirih uporabnih kompozitnih komponent: komponente za vnos datuma, katera z uporabo pomožne komponente vnesen datum neposredno pretvori v javanski objekt Date, komponente spinner, ki omogoča vnos števila znotraj določenih mej, komponente za registracijo, ki vsebuje pogosto prisotna polja za vnos uporabnikovih podatkov pri registraciji, funkcijo za validiranje gesla in je načrtovana po principu tekočega izgleda, in komponente za samodokončanje vnosa, ki s pomočjo tehnologije AJAX uporabniku ob tipkanju takoj ponudi možne izbire.

Ključne besede: kompozitne komponente, JSF, JavaServer Faces, spletne aplikacije, odzivnost.

Abstract

Due to the increase in the number of mobile devices, building responsive applications is currently one of the biggest challenges in web application development. In the sphere of software development principles there is an emphasis on development of modular components that are reusable. The thesis describes the development of composite components in JavaServer Faces (JSF) technology in detail, which we develop independent components in, that are fully reusable and can be included in any JSF web application. In addition, the components are responsive and thereby relieve developers from this unpleasant work. At the beginning, the thesis describes operation of JSF technology and development of composite components in detail, followed by a description of the technologies used to achieve responsiveness. Next chapter displays the development of four useful composite components in detail: a component for entering the date, which directly converts entered date into Java Date object through use of a backing component, a spinner component, which allows entering numbers within certain boundaries, a registration component, containing input fields, which are often present in the process of user registration, a function for validating passwords and uses a fluid design, and an autocomplete component, which uses AJAX technology to immediately provide possible choices to the user during typing.

Keywords: composite components, JSF, JavaServer Faces, web applications, responsiveness.

Poglavje 1

Uvod

Izdelava obogatenih spletnih aplikacij (ang. *rich internet applications - RIA*) v današnjem času doživlja velik razvoj. Uporabniki kažejo vedno večjo potrebo po tem, da spletne aplikacije izgledajo in delujejo kot običajne aplikacije na računalnikih. Hkrati pa se trenutno kaže porast uporabe mobilnih naprav za dostopanje do spletnih aplikacij, za katere uporabniki želijo, da se obnašajo in izgledajo tako dobro kot na osebnih računalnikih. Zato je vsak dan na voljo vedno več orodij in tehnologij, ki nam omogočajo izdelavo odzivnih spletnih aplikacij. Na področju samega postopka razvoja aplikacij nasploh pa se kaže težnja k čim hitrejšemu razvoju brez ponavljanja kode in modularnosti samih aplikacij. Ena izmed možnosti, da na področju spletnih aplikacij izpolnimo zgornje težnje in sledimo zadnjim smernicam, je razvoj odzivnih kompozitnih komponent v tehnologiji JSF. Tako ustvarimo komponente, ki enkapsulirajo določeno funkcionalnost in jih razvijalci spletnih strani le vključijo v aplikacijo, s čimer se njen razvoj pospeši. Hkrati pa so komponente z ustreznim pristopom pri implementaciji odzivne in izpolnijo pričakovanja uporabnikov, ki uporabljajo različne naprave.

Namen diplomske naloge je analizirati in proučiti način razvoja kompozitnih komponent v javanski spletni tehnologiji JSF in razviti štiri nove kompozitne komponente. Skozi primere bomo skušali prikazati, kakšne so prednosti razvoja kompozitnih komponent in zakaj je tak pristop boljši kot pisanje ponavljajoče se kode skozi več aplikacij. Prikazali bomo, kako je moč doseči odzivnost komponent, tako da izgledajo žive, se takoj odzivajo na želje oz. ukaze uporabnika in se prilagajajo uporabnikovi napravi. Cilj diplomske naloge je najprej predstaviti uporabljene

tehnologije in razviti štiri primere uporabnih kompozitnih komponent, ki bodo realizirane tako, da bodo vsebovale čim več smiselnih funkcionalnosti in bodo čim bolj izpopolnjene. Cilj je tudi razviti komponente, ki bodo že same po sebi prilagodljive in odzivne uporabniku in za njihovo uporabo ne bodo potrebne dodatne nastavitve. Bralcu bomo torej pokazali zgoraj omenjene prednosti razvoja komponent in kaj vse lahko realiziramo s samimi komponentami.

V diplomski nalogi bomo najprej predstavili strežniško tehnologijo JSF, ki jo bomo uporabljali kot podlago pri razvoju komponent. V naslednjem poglavju bomo podrobno predstavili kompozitne komponente, kako jih razvijemo in kakšne funkcionalnosti lahko vsebujejo. Nato bomo predstavili uporabljane tehnologije na strani odjemalca, s katerimi bomo dosegli odzivnost kompozitnih komponent. Nato bo sledil podroben prikaz razvoja štirih v praksi uporabnih odzivnih komponent, preko katerega bo bralec lahko spoznal, kako raznovrstne in funkcionalne komponente je moč razviti. Na koncu bo sledil zaključek z glavnimi spoznanji, s katerimi smo se srečali pri izdelavi kompozitnih spletnih aplikacij.

Poglavje 2

Pregled tehnologije JSF

JavaServer Faces (v nadaljevanju JSF) je strežniško (ang. *server-side*) ogrodje za razvoj spletnega predstavitvenega sloja oz. uporabniškega vmesnika aplikacij Java EE [32, 12, 31]. Razvit je bil kot odgovor na razvijalcem neprijazni obstoječi javanski ogrodji za razvoj spletnih aplikacij JavaServer Pages (JSP) in Java Servlets. Pomembno je omeniti, da je sam JSF le specifikacija, ki ima različne implementacije (npr. Mojarra). Tudi samo ogrodje JSF se je od začetne verzije izpopolnjevalo (bolj hiter in razvijalcem prijazen razvoj, podpora tehnologijam AJAX in REST itd.), v času pisanja je v uporabi verzija 2.2 (Mojarra)[1].

Glavna izboljšava oz. prednost JSF v primerjavi s starejšimi javanskimi ogrodji za razvoj spletnih aplikacij (npr. JSP in Java Servlet API) sta MVC [26] in komponentni model, ki omogočata povezavo komponent JSF z javanskimi objekti v ozadju brez mešanja programske in predstavitvene (ang. *markup*) kode. To posledično prinese boljšo integracijo s poslovno logiko (EJB) in podatkovnim modelom (JPA). JSF vsebuje vso potrebno kodo za obravnavanje dogodkov in organizacijo komponent (preslikava itd.), tako da se programerji lahko posvetijo sami aplikacijski logiki. Prednost tehnologije JSF je tudi preprosta vključitev raznih odprtokodnih razširitev, kot so Seam, RichFaces, Pretty Faces in podobne [1].

[1] https://javaserverfaces.java.net

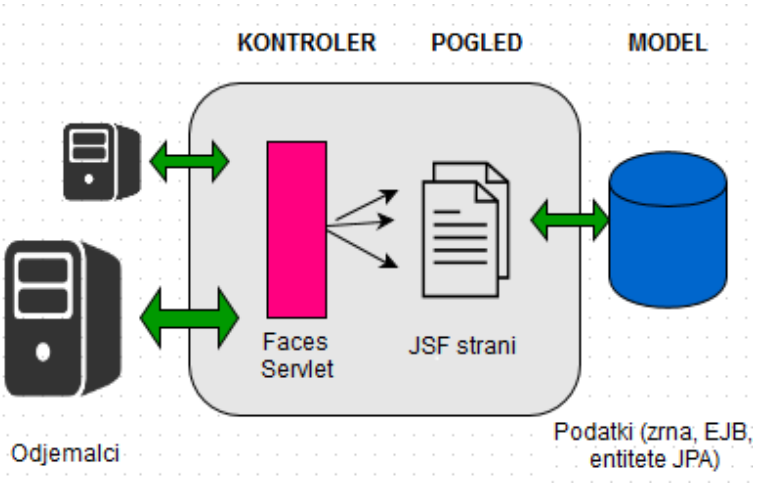
Kot glavne sestavne dele JSF bi torej lahko navedli [32]:

- množico komponent JSF za opis uporabniškega vmesnika (ang. *UI*)
- dogodkovno vodeni programski model
- komponentni model, ki omogoča vključitev še raznih zunanjih (ang. *third-party*) komponent

JSF bazira na komponentah (ang. *component-based*), saj razvijalec izgled opiše s pomočjo komponent JSF, ki se ob zahtevku HTTP izvedejo na strani strežnika in ustrezno preslikajo v končni HTML, katerega strežnik vrne kot odgovor. JSF za opis izgleda aplikacij uporablja datoteke XML (t.i. Facelets oz. View templates). Ob prejeti zahtevi HTTP FacesServlet procesira zahtevo, naloži ustrezen View template, ustvari komponentno drevo (ang. *component tree*), procesira dogodke in generira odgovor odjemalcu v jeziku HTML [31]. Stanje komponent in ostalih objektov v ustreznem območju (ang. *scope*) se shrani ob koncu vsake zahteve v procesu `stateSaving` in se obnovi ob naslednjem kreiranju tega pogleda. Objekte in stanje lahko shranjujeta tako odjemalec kot strežnik. Komponentni razredi JSF UI torej enkapsulirajo funkcionalnost komponent, ne pa njihove predstavitve pri odjemalcu, kar omogoča, da so komponente JSF lahko izrisane na različnih napravah (odzivne in fleksibilne). S kombinacijo funkcionalnosti komponent in izdelave lastnih izrisovalnikov (ang. *renderers*) lahko razvijalci izdelajo svoje značke za določene odjemalce.

Model MVC v JSF sestavljajo (glej sliko 2.1) [31, 21]:

- podatki in omejitve, vezane na njih - zrna, entitete JPA in EJB (Model)
- uporabniški vmesnik – strani JSF z uporabo komponentnega modela, npr. UIComponents (Pogled)
- interakcija med up. vmesnikom in podatki – npr. FacesServlet (Kontroler)



Slika 2.1: MVC v domeni tehnologije JSF [31, 21]

Primer osnovne aplikacije JSF po njenih sestavnih delih:

1. Izgled (ang. *design*): JSF View template oz. stran JSF (izsek kode 2.1)

Izsek kode 2.1: Stran JSF

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>Pozdravljeni</title>
   </h:head>
   <h:body>
      <h:form>
         <h3>Vpisite ime:</h3>
            <h:inputText value="#{uporabnik.ime}"/>
              <p><h:commandButton value="Prijava" action="pozdravljeni"/></p>
      </h:form>
   </h:body>
</html>
```

Stran JSF oz. View template je datoteka XHTML oz. XML z dodanimi značkami JSF iz ustreznih imenskih prostorov XML. V tem primeru zasledimo naslednje lastnosti strani JSF [31]:

- za uporabo osnovnih komponent HTML v JSF moramo definirati naslovni prostor **h**, ki kaže na dogovorjen URL
- komponente JSF uporabimo tako, da jim pred imenom navedemo naslovni prostor JSF, v katerem so definirane
- zapis `#{...}` pomeni uporabo izraznega jezika (ang. *expression language*)
- polja za vnos so povezana z lastnostmi objekta (`#{uporabnik.ime}`), ki je definiran v ustreznem javanskem zrnu
- ob kliku na gumb se sproži preusmeritev na stran `pozdravljeni.xhtml`

Datoteke JSF morajo biti pravilno formatirane, saj implementacija JSF ne dovoli sintaktičnih napak, in kjer lahko, raje uporabimo komponente JSF kot standardne značke HTML. Značke JSF (ang. *JSF tags*) v grobem delimo na [31]:

- značke Core (f): nastavljanje atributov, parametrov, validatorjev, poslušalcev, pretvornikov itd.
- značke HTML (h): ustrezajo značkam HTML z istim pomenom
- značke Facelets: implementacija pogleda z uporabo Faceletsov, ustvarjanje in razširjanje predlog, razni pripomočki itd.
- značke Composite: izdelava kompozitnih komponent
- značke JSTL Core: uporaba zank, vejitev, pogojnih izrazov itd.
- značke JSTL Functions: vključujejo razne uporabne funkcije za uporabo na strani JSF, predvsem za delo z nizi

Vsakemu razredu značk pripada ustrezen naslovni prostor XML, ki ga moramo vključiti v stran JSF, da jih lahko uporabljamo, in ob uporabi le-teh navesti ustrezen akronim za naslovni prostor XML.

2. Aplikacijska koda: javansko zrno (ang. *Java bean*), ki upravlja s podatki o uporabniku - izsek 2.2

Izsek kode 2.2: Zrno UporabnikZrno.java

```
import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean(name="uporabnik")
@RequestScoped
public class UporabnikZrno implements Serializable {
   private String ime;
   public String getIme() { return ime; }
   public void setIme(String novaVrednost) { ime = novaVrednost; }
}
```

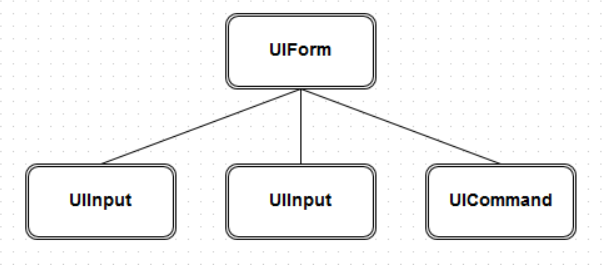
Zrno je javanski razred, ki izpostavi lastnosti objekta in dogodke ogrodju z uporabo konvencij za poimenovanje in metod za nastavljanje ter vračanje vrednosti (ang. *getter and setter methods*) [31]. V aplikacijah JSF se uporablja t.i. upravljano zrno (ang. *managed bean*), do katerega lahko dostopajo strani JSF. JSF 2.2 nam omogoča preprosto nastavitev razreda z uporabo javanskih anotacij. Z anotacijo `@ManagedBean` deklariramo javansko zrno z ustreznim imenom, z anotacijo `@RequestScoped` pa omogočimo hranjenje stanja objekta le za posamezno zahtevo HTTP (alternative so `@SessionScoped`, kjer se objekt hrani celotno sejo; `@ApplicationScoped`, kjer se objekt hrani celoten življenjski čas aplikacije; `@ConversationScoped`, kjer se objekt hrani čez več izvedb življenjskega cikla JSF oz. množice povezanih strani, ki jih je določil razvijalec; in `@ViewScoped`, kjer se stanje objekta hrani, dokler se uporabnik nahaja na isti strani [24]). Zrna so upravljana v tem smislu, da ob pojavitvi imena zrna v strani implementacija JSF locira objekt s tem imenom in ga konstruira (če že ne obstaja) v določenem območju vidnosti (ang. *scope*). V novejših verzijah Jave EE (6 in naprej) pa lahko namesto upravljanih zrn uporabimo zrna CDI, ki delujejo po principu CDI (Context Dependecy Injection) [6, 31] – namesto uporabe anotacije `@ManagedBean` uporabimo anotacijo `@Named`. Prednost zrn CDI v primerjavi z upravljanimi zrni je bolj fleksibilen model, saj so omejena na nek kontekst (ang. *context*), katerega lahko tudi sami definiramo. Poleg tega CDI tudi specificira mehanizme za vstavljanje zrn, prestrezanje klicev metod, proženje dogodkov itd. V aplikacijah JSF se torej zrna uporabljajo kot povezava med predstavitvenim slojem (ang. *UI*) in zaledjem aplikacije (ang. *backend*).

Ko bo stran JSF, prikazana v izseku kode 2.1, naložena, se bo poklicala metoda `getIme()`, ki bo pridobila trenutno stanje objekta. Ko bo stran v poslanem stanju (ang. *submitted*), pa se bo poklicala metoda `setIme()`, ki bo zrnu nastavila vrednost, ki je bila vnesena v formo.

Aplikacija JSF uporablja tudi konfiguracijske datoteke. Datoteka `web.xml` je deskriptorska datoteka XML, ki opisuje servlete in ostale komponente, ki sestavljajo našo aplikacijo. V njej nastavimo vstopno stran in `FacesServlet` na želeni naslov URL, na kateremu se bo odzivala naša aplikacija. Datoteka `faces-config.xml` pa vsebuje konfiguracijo samega JSF, v kateri lahko deklariramo navigacijska pravila, konfiguriramo zrna, validatorje, pretvornike, poslušalce faznih dogodkov itd. Vendar v JSF 2.2 zaradi preprostosti namesto konfiguracije v datoteki raje uporabimo konfiguracijo z anotacijami, kjer je to mogoče.

2.1 Življenjski cikel aplikacije JSF

Ob dostopu do vstopne strani aplikacije preko brskalnika (vzemimo za primer datoteko `index.xhtml`) JSF inicializira kodo JSF in prebere stran [31]. Stran vsebuje značke JSF (npr. `h:form`, `h:inputText` itd.) in vsaka značka ima dodeljen razred za obravnavanje značk (ang. *tag handler*). Ko je stran prebrana, se razredi za obravnavanje značk zaženejo in sodelujejo med seboj pri izgradnji komponentnega drevesa. Komponentno drevo je podatkovna struktura, ki vsebuje javanske objekte za vse predstavitvene elemente na strani JSF (slika 2.2).



Slika 2.2: Primer komponentnega drevesa za stran JSF z dvema vnosnima poljema in gumbom

2.1.1 Izris strani JSF

V naslednjem koraku je stran JSF izrisana. Značke se preslikajo v ustrezne elemente HTML z ustreznimi atributi [13]. Ob vsaki znački se ustvari tudi komponenta, ki ji ustreza. Vsaka komponenta ima svoj izrisovalnik (ang. *renderer*), ki proizvede kodo HTML, ki vsebuje ustrezne atribute in ponazarja stanje komponente. Značka `<h:inputText value="uporabnik.ime"/>` se kot primer preslika v značko

```
<input type="text" name="unikaten ID" value="vrednost"/>
```

z ustreznimi atributi. Ta proces se imenuje kodiranje (ang. *encoding*). Izrisovalnik objekta UIInput pozove implementacijo JSF, da pogleda njegov unikatni identifikator (privzeto jih dodeli JSF) in vrednost izraza `uporabnik.ime`. Kodirana stran je nato poslana brskalniku, ki jo prikaže.

2.1.2 Dekodiranje zahtev

Ko je stran prikazana v brskalniku in uporabnik vnese podatke ter klikne gumb za pošiljanje forme, brskalnik pošlje podatke forme (ang. *form data*) strežniku, formatirane kot zahteva POST [31]. Podatki forme se dodajo v razpršeno tabelo in so dostopni vsem komponentam. Nato implementacija JSF da priložnost vsaki komponenti, da pregleda razpršeno tabelo. Ta proces se imenuje dekodiranje (ang. *decoding*). Vsaka komponenta se lahko po svoje odloči, kako bo interpretirala te podatke. Če vzamemo za primer formo za prijavo z dvema vnosnima poljema (dve komponenti UIInput) in gumbom (komponenta UICommand), se proces izvrši v naslednjem redu:

- komponenta UIInput posodobi lastnosti zrna, na katere se nanašajo atributi `value` – izvedejo se metode za nastavljanje vrednosti (ang. *setters*) z vrednostmi, ki jih je podal uporabnik
- komponenta UICommand preverja, če je bil kliknjen gumb. Če je bil, sproži dogodek (ang. *action event*), ki je referenciran v atributu `action`

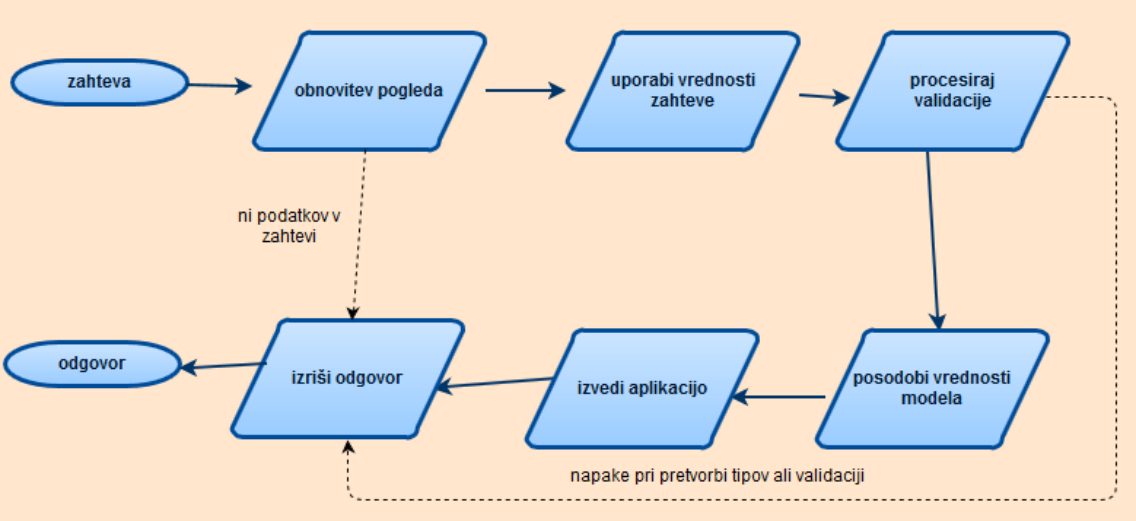
2.1.3 Faze življenjskega cikla JSF

JSF specifikacija definira šest faz [20, 31]:

1. obnovitev pogleda
2. uporabi vrednosti zahteve
3. procesiraj validacije
4. posodobi vrednosti modela
5. izvedi aplikacijo
6. izriši odgovor

Prva faza pridobi komponentno drevo za zahtevano stran (če je bila že prej prikazana) ali pa konstruira novo v primeru, da je bila stran prikazana prvič. Če zahteva nima podatkov (ang. *request value data*), implementacija JSF preskoči v zadnji korak (to se najpogosteje zgodi pri prvem prikazu strani). V nasprotnem primeru pa preide v drugo fazo. V tej fazi JSF iterira po komponentnih objektih v komponentnem drevesu. Vsaka komponenta preverja, če ji vrednosti iz zahteve pripadajo, in jih v tem primeru shrani. Vrednosti, shranjene v komponenti, se imenujejo lokalne vrednosti. V tej fazi se zgodi tudi pretvorba vhodnih podatkov, ki so predstavljeni kot zaporedja znakov (ang. *string*), v ustrezne podatkovne tipe. V komponentah so tako vrednosti shranjene kot ustrezni podatkovni tipi (int, Date itd.). Če so bili ob izgradnji strani JSF dodani tudi validatorji, se le-ti izvedejo v naslednji, tretji fazi. Če je validacija uspešna, se življenjski cikel nadaljuje naprej. V primeru, da se pojavijo napake ob validaciji ali pretvorbi tipov, pa JSF izvede zadnjo fazo, v kateri ponovno prikaže trenutno stran in opis napake. Po uspešni validaciji se lahko varno posodobijo vrednosti modela (četrta faza). V tej fazi se lokalne vrednosti uporabijo za posodobitev zrn, s katerimi so komponente povezane. Podatki se v modelu ne posodobijo kar takoj (brez izvedbe druge in tretje faze) ravno zaradi validacije, saj v primeru napak oz. neveljavnih vnosov podatkov v modelu ne spreminjamo. Nato nastopi peta faza, v kateri se izvede akcijska metoda (ang. *action method*) komponente gumba ali povezave, ki je sprožila pošiljanje forme. V tej metodi se lahko izvede tudi drugo aplikacijsko procesiranje. Metoda vrne objekt String, ki se pošlje upravljavcu navigacije (ang.

navigation handler), ki pridobi naslednjo stran (objekt String se na podlagi navigacijskih pravil pretvori v ustrezen identifikator strani (ang. *view ID*)). Na koncu se v šesti fazi kodira odgovor in pošlje brskalniku odjemalca. Celoten življenjski cikel JSF je prikazan na sliki 2.3.



Slika 2.3: Celoten življenjski cikel JSF [31]

2.2 Procesiranje dogodkov v JSF

JSF podpira naslednje tipe dogodkov [31]:

- dogodki ob spremembi vrednosti (ang. *value change events*) – pri vnosnih poljih
- akcijski dogodki (ang. *action events*) – klik na gumb ali povezavo
- fazni dogodki (ang. *phase events*) – rutinsko se izvajajo v ciklu JSF
- sistemski dogodki (ang. *system events*)

Vsi ti dogodki se izvedejo na strani strežnika in ne na strani odjemalca. Zahteve implementacija JSF procesira z razredom `ControllerServlet`, ki izvaja cikel JSF. Implementacija JSF lahko, začenši z drugo fazo, ustvarja dogodke in jih dodaja v dogodkovno vrsto v vsaki fazi cikla. Po vsaki fazi implementacija JSF oddaja dogodke v vrsti registriranim poslušalcem.

2.2.1 Dogodki ob spremembi vrednosti

Ti dogodki se nanašajo na spremembo vrednosti pri vnosnih poljih na strani JSF [31]. Na vnosno polje registriramo rokovalnik dogodka z atributom `valueChangeListener`, ki mu kot vrednost damo referenco na ustrezno metodo, ki implementira poslušalca. Pri tem je potrebno poudariti, da samo registracija metode ne bo ob spremembi vrednosti izvršila ničesar, saj se življenjski cikel JSF ne izvede samodejno. To rešimo z uporabo tehnologije AJAX ali z registracijo poslušalca dogodka `onchange=submit()` in tako pošljemo formo ter zaženemo življenjski cikel.

2.2.2 Akcijski dogodki

Akcijske dogodke povzročijo kliki na gumb ali povezavo [16, 31]. Natančneje se ti dogodki sprožijo v predzadnji fazi življenjskega cikla JSF. Poslušalca lahko registriramo na t.i. akcijsko komponento z atributom `actionListener`. Lahko pa uporabimo tudi značko `f:actionListener` (enako velja tudi za prejšnjo vrsto dogodkov), ki nam omogoča registracijo več poslušalcev na isto komponento.

2.2.3 Fazni dogodki

JSF proži fazne dogodke po vsaki fazi in pred njo [31]. Te dogodke obravnavajo t.i. fazni rokovalniki (ang. *phase handlers*). Fazne dogodke implementiramo z razširitvijo vmesnika `PhaseListener`, kjer implementiramo metode, ki se izvedejo po vsaki fazi in pred njo. Lahko pa jih obravnavamo tudi z uporabo ustreznih značk JSF. Ti dogodki nam največkrat služijo za pomoč pri razhroščevanju aplikacije (ang. *debug*).

2.2.4 Sistemski dogodki

JSF 2.2 uvaja finozrnat (ang. *fine-grained*) sporočilni sistem, v katerem tako individualne komponente kot implementacija JSF obvestijo poslušalce raznih potencialno zanimivih dogodkov, npr. pred zagonom ali koncem aplikacije, izrisom ali odstranitvijo komponente itd [16]. Nek razred lahko pridobi sistemske dogodke z uporabo značke `f:event`, z uporabo anotacij, nastavitve konfiguracijske datoteke

ali klicem ustrezne metode razreda `UIComponent`. Uporaba sistemskih dogodkov nam omogoča tudi validacijo skupine komponent.

2.3 Razvoj uporabniškega vmesnika z značkami JSF Facelets

Ker v delu obravnavamo razvoj po komponentah oz. modularen razvoj, je smiselno predstaviti tudi JSF Facelets. Facelets so v JSF vključeni od verzije 2.0 naprej, da bi omogočili hitrejši razvoj bolj fleksiblinega uporabniškega vmesnika aplikacij JSF [32, 31, 9]. Facelets uvaja nove značke Facelets (ang. *Facelets Tags*), ki jih lahko razdelimo v naslednje skupine (uporabljajo predpono `ui`) [31]:

- značke za vstavljanje vsebine iz drugih datotek XHTML (ang. *templating*): npr. `ui:include`
- značke za gradnjo strani iz predlog (ang. *templates*): npr. `ui:insert`, `ui:define`, `ui:composition`
- značke za izgradnjo lastnih komponent brez pisanja javanske kode: npr. `ui:component`
- razni pripomočki: npr. `ui:repeat`, `ui:debug`

Podrobnejši opis značk Facelets se nahaja v tabeli 2.1 [31].

Tabela 2.1: Značke Facelets

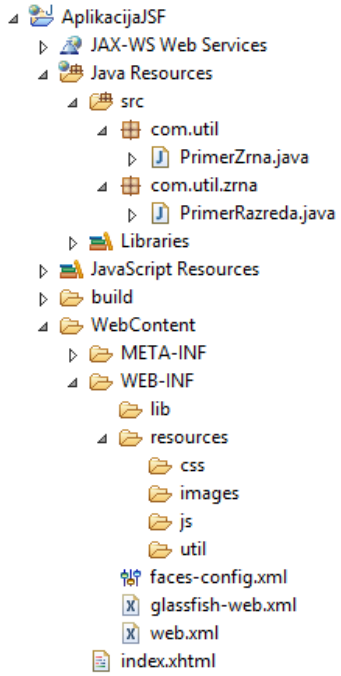
Značka	**Opis**
`ui:component`	Ustvari komponento, ki je dodana v komponentno drevo.
`ui:composition`	Brez uporabe atributa `template` značka označuje sekvenco elementov, ki bodo lahko vstavljeni nekam drugam. Če pa značka uporablja še atribut `template`, pomeni nalaganje ustrezne predloge.
`ui:debug`	Omogoča prikaz okna za razhroščevanje ob pritisku na tipko, določeno z atributom `hotkey`.
`ui:decorate`	Identična znački `ui:composition`, le da ne spregleda vsebine strani izven nje.
`ui:define`	Definira vsebino, ki bo vnesena v mesto v predlogi z imenom, ki ga določa značka `ui:insert` z isto vrednostjo atributa `name`.
`ui:fragment`	Identično kot `ui:component`, le da ne spregleda vsebine izven nje.
`ui:include`	Vstavi vsebino iz druge datoteke XHTML, določene z atributom `src`.
`ui:insert`	Uporablja se v predlogi in definira mesto, kamor bo vnesena vsebina, različna od strani do strani. Uporablja atribut `name`, s katerim poimenujemo posamezen odsek.
`ui:param`	Specificira parameter, ki je posredovan vključeni datoteki. Atribut `name` določa ime parametra, atribut `value` pa njegovo vrednost.
`ui:remove`	JSF odstrani vso vsebino, ki se nahaja znotraj te značke.
`ui:repeat`	Iterira čez seznam, polje, množico ali posamezen objekt, določen z atributom `value` s korakom, določenim z atributom `step`. Trenutna vrednost se hrani v spremenljivki, določeni z atributom `var`.

Facelets nam tako omogoča, da bi v primeru potrebe po spremembi aplikacije namesto vsake strani posebej spremenili le njeno predlogo, kar bi nam kot razvijalcem prihranilo precej časa. Uporabni pa so seveda tudi v primeru (ki ga srečamo pri večini spletnih aplikacij), ko ima naša aplikacija nek načrt izgleda (ang. *layout*), ki se ponavlja na vseh straneh in bi nam ga bilo brez uporabe Faceletsov potrebno ponavljati na vsaki strani posebej. Facelets tako sledi principu DRY (Don't Repeat Yourself), ki ga poznamo v večini novejših tehnologij za razvoj spletnih aplikacij (Django, Ruby On Rails itd.) [5]. Aplikacija, ki uporablja Facelets je tako sestavljena iz glavne predloge (ang. *master template*), ki vsebuje osnoven načrt strani in t.i. odjemalskih predlog (ang. *template clients)*, ki jo razširjajo. Seveda lahko ustvarimo tudi hierarhijo predlog. Če torej hočemo, da bo naša aplikacija uporabljala Facelets za pogled, moramo ustrezno nastaviti `FacesServlet` na naslov URL naše aplikacije v datoteki `web.xml` in po potrebi še specificirati dodatno konfiguracijo glede navigacije, zrn itd. v datoteki `faces-config.xml`.

Facelets nam omogoča ustvarjanje lastnih značk, ki so lahko združene iz več drugih značk JSF in jih tako le vstavimo v stran, pri čemer jih moramo ustrezno deklarirati v knjižnici značk (ang. *tag library*) v ustreznem imenskem prostoru. Bolj napreden mehanizem od lastnih značk so kompozitne komponente, ki nam omogočajo še dodajanje funkcionalnosti značkam.

2.4 Struktura spletnega projekta, ki uporablja JSF

Za lažje razumevanje razvoja kompozitnih komponent bomo v tem podpoglavju predstavili strukturo projekta, ki uporablja JSF. Slika 2.4 prikazuje strukturo projekta v okolju Eclipse z osnovno aplikacijo JSF [19].



Slika 2.4: Struktura projekta z osnovno aplikacijo JSF

Direktorij *WebContent* predstavlja korensko (ang. *root*) mapo z elementi spletne aplikacije [19]. V njej se nahajajo strani JSF in mapi *META-INF* ter *WEB-INF*. Mapa *META-INF* vsebuje razne metapodatke aplikacije. Bolj pomembna je mapa *WEB-INF*, ki vsebuje sestavne dele spletne aplikacije - konfiguracijske datoteke `web.xml`, `faces-config.xml` in `glassfish-web.xml` (slednjo naš projekt vsebuje, ker uporabljamo strežnik Glassfish in vsebuje dodatne nastavitve zanj) ter razne druge mape. V mapo *lib* vstavimo zunanje .jar datoteke, v mapo *resources* pa druge sestavne dele spletne aplikacije, ki jih potrebuje za njeno delovanje. V mapi *WEB-INF* se v primeru, da uporabljamo predloge, nahaja tudi mapa *templates*, ki jih vsebuje. V našem primeru bomo v mapi *resources* imeli razne zunanje knjižnice (Boostrap, Prototype), datoteke JavaScript (mapa *js*) in kompozitne komponente v mapi, ki ustreza njenemu akronimu naslovnega prostora XML (v našem primeru se bodo nahajale v mapi *util*). V projektu pa bomo uporabljali tudi javanske datoteke, ki jih bomo imeli v mapi *src* v ustreznem javanskem paketu. V paketu *com.util.zrna* se bodo nahajala uporabljena zrna, v paketu *com.util* pa ostali javanski razredi.

Poglavje 3

Kompozitne komponente JSF

Kot je bilo predstavljeno v prejšnjem poglavju, JSF bazira na komponentah (ang. *component-based*), kar pomeni, da je mogoče implementirati komponente, ki jih lahko ponovno uporabimo. Komponente so tako dober mehanizem za ponovno uporabo. Ker jih JSF 1.0 ni omogočal implementirati na preprost način, so avtorji ogrodja z verzijo JSF 2.0 poenostavili izgradnjo novih komponent v javanski kodi in kombiniranje že obstoječih komponent v eno komponento, ki je na preprost način ponovno uporabljiva. Tako dobimo t.i. kompozitne komponente [32, 31, 29].

3.1 Knjižnica značk JSF Composite Tag Library

JSF 2.2 uporablja knjižnico značk za implementacijo kompozitnih komponent [32, 31]. Po konvenciji se zanjo uporablja predpona `composite`. Za njeno uporabo moramo deklarirati imenski prostor XML:

`xmlns:composite="http://java.sun.com/jsf/composite".`

Opis kompozitnih značk se nahaja v tabeli 3.1 [31, 32].

Tabela 3.1: Kompozitne značke `composite`

Značka	**Opis**
`actionSource`	Izpostavi komponento, ki proži akcijske dogodke.
`attribute`	Izpostavi atribut komponente.
`editableValueHolder`	Izpostavi komponento, ki hrani spremenljivo vrednost.
`extension`	Značka lahko vsebuje samostojen XML.
`facet`	Deklarira podporo značke `facet` v tej komponenti z danim imenom.
`implementation`	Vsebuje zapis XHTML, ki definira komponento.
`insertChildren`	Vstavi komponente kot otroke, ki jih je specificiral avtor strani.
`insertFacet`	Vstavi facet (vsebino XHTML), ki ga je specificiral avtor.
`interface`	Vsebuje druge kompozitne značke, ki izpostavljajo akcijske vire, atribute in druge podkomponente kompozitne komponente.
`renderFacet`	Izriše facet, ki ga je specificiral avtor strani.
`valueHolder`	Izpostavi komponento, ki hrani vrednost.

Kompozitne komponente so sestavljene iz vmesnika in implementacije. Implementacija je koda Facelet oz. HTML, sestavljena iz standardnih značk JSF. Vmesniki pa nam omogočajo izpostaviti konfigurabilne značilnosti naše kompozitne komponente.

3.2 Uporaba kompozitnih komponent

V želji po odpravi za razvijalce obremenjujoče kofiguracije (pravilo konvencija pred konfiguracijo) JSF 2.2 uporablja razne konvencije glede imen [32, 31]. Če hočemo uporabiti kompozitno komponento `<util:debug />`, ki nam izpiše podatke o prejeti zahtevi HTTP, moramo deklarirati ustrezen imenski prostor na naslednji način:

```
xmlns:util="http://java.sun.com/jsf/composite/util".
```

Zadnji del poti (`util`) predstavlja direktorij *util*, ki se nahaja v mapi *resources* (torej *resources/util*). V direktoriju *util* pa se nahaja datoteka z opisom kompozitne komponente `debug`. Vzemimo primer preproste kompozitne komponente `util:ikona`, na kateri s klikom sprožimo nek dogodek. V strani JSF jo želimo uporabiti na način, kot kaže izsek kode 3.1:

Izsek kode 3.1: Uporaba kompozitne komponente `ikona`

```
<util:ikona slika="#{resource['images:prijava.jpg']}"
          metoda="#{uporabnik.prijava}" />
```

Kot vsaka kompozitna komponenta bo tudi naša sestavljena iz implementacije in vmesnika, ki nam bo omogočal njeno ponovno (večkratno) uporabo pod različnimi pogoji – to je zagotovljeno z uporabo njenih atributov `slika` in `metoda`. Komponenta je definirana v datoteki `ikona.xhtml` (izsek 3.2).

Izsek kode 3.2: Datoteka `ikona.xhtml` z definicijo komponente

```
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:composite="http://java.sun.com/jsf/composite">

  <composite:interface>
     <composite:attribute name="slika"/>
     <composite:attribute name="metoda"
          method-signature="java.lang.String action()" />
  </composite:interface>
  <composite:implementation>
     <h:form>
        <h:commandLink action="#{cc.attrs.metoda}">
        <h:graphicImage url="#{cc.attrs.slika}" />
        </h:commandLink>
     </h:form>
   </composite:implementation>
</html>
```

V implementaciji do posameznega atributa dostopamo z izrazom `#{cc.attrs.attrname}`, pri čemer `attrname` predstavlja ime atributa. `cc` predstavlja komponento, `cc.attrs` pa vse atribute te komponente. Privzeto JSF razume vse vrednosti atributov kot tip `java.lang.Object`. V našem primeru se pri uporabi prvega atributa njegova vrednost spremeni v ustrezen niz URL. Če želimo, da atribut predstavlja podrazred razreda `java.lang.Object`, potem mu moramo

definirati tip. To lahko naredimo z atributom `method-signature` ali z atributom `type`. V izseku 3.3 je prikazan primer atributa `datum`, ki sprejme javanski objekt Date in zato mora imeti ustrezno definiran tip z atributom `type`.

Izsek kode 3.3: Uporaba atributa `type`

```
<composite:attribute name="datum" type="java.util.Date" />
```

Posamezen atribut v vmesniku lahko nastavimo na zahtevan, dodamo atribute, ki nam omogočajo validacijo, nastavitev razreda CSS, itd. [32, 31]. Pri opisu kompozitnih komponent pa moramo izpostaviti še njihovo glavno lastnost (ki jih tudi ločuje od sestavljenih značk, omenjenih v prejšnjem poglavju) – **interakcija s podatki na strani strežnika**. Delovanje bomo razložili na primeru uporabne kompozitne komponente v obliki obrazca za prijavo, ki vsebuje polje za vnos uporabniškega imena, gesla in gumb. Uporabiti jo želimo na način, prikazan v izseku 3.4.

Izsek kode 3.4: Uporaba komponente za prijavo

```
<util:prijava metoda="#{uporabnik.prijava}"
    gumbOpis="#{uporabnik.gumbOpis}"
    uporabnik="#{uporabnik}"/>
```

Kot vrednost atributa `uporabnik` mora avtor strani JSF (uporabnik komponente) vnesti referenco na javansko zrno `uporabnik` in ustrezne druge vrednosti atributov. Komponento lahko implementiramo na način, kot kaže izsek 3.5.

Izsek kode 3.5: Definicija komponente za prijavo

```
<composite:interface>
  <composite:attribute name="uporabnik"/>
  <composite:attribute name="gumbOpis"/>
  <composite:attribute name="metoda"
    method-signature="java.lang.String action()"/>
</composite:interface>
<composite:implementation>
  <h:form id="forma">
    <h:panelGrid columns="2">
    Up. Ime:
      <h:inputText id="ime" value="#{cc.attrs.uporabnik.ime}"/>
    Geslo:
      <h:inputSecret id="geslo" value="#{cc.attrs.uporabnik.geslo}"/>
    </h:panelGrid>
```

```
        <p>
          <h:commandButton id="gumbPrijava"
            value="#{cc.attrs.gumbOpis}"
            action="#{cc.attrs.metoda}"/>
        </p>
      </h:form>
    </composite:implementation>
```

Podajanje reference na zrno preko atributa je ena izmed strategij za interakcijo s strežnikom. Vendar bo, kot je razvidno, na ta način komponenta delovala le z zrni, ki imajo lastnosti, poimenovane z `ime` in `geslo`. Da bi omogočili komponenti, da bi bila bolj uporabna (z različnimi zrni), uporabimo finozrnat pristop, in sicer lahko deklariramo še atributa `ime` in `geslo`, katerih vrednosti bo vnesel uporabnik komponente glede na to, kako so poimenovane v njegovem zrnu. Uporabnost komponente lahko še povečamo z omogočanjem uporabe validatorjev [25]. To lahko na našem primeru omogočimo z nekaj popravki v kodi. V vmesnik komponente dodamo tri značke `composite:editableValueHolder`, prva se nanaša na `ime`, druga na `geslo` in tretja na oboje (izsek 3.6).

Izsek kode 3.6: Vmesnik komponente z značkami za izpostavitev podkomponent

```
<composite:interface>
  <composite:editableValueHolder name="imeVnos"
    targets="forma:ime"/>
  <composite:editableValueHolder name="gesloVnos"
    targets="forma:geslo"/>
  <composite:editableValueHolder name="vnosi"
    targets="forma:ime forma:geslo"/>
  ...
</composite:interface>
```

Tako uporabniku komponente izpostavimo njene podkomponente, referencirane z atributom `targets` pod imenom, definiranim z atributom `name`. Primer dodajanja validatorjev na komponento je prikazan v izseku 3.7.

Izsek kode 3.7: Uporaba validatorjev pri komponenti za prijavo

```
<util:prijava ...>
  <f:validateLength maximum="10" for="vnosi"/>
  <f:validateLength minimum="4" for="imeVnos"/>
  <f:validator id="com.util.Geslo" for="gesloVnos"/>
```

```
</util:prijava>
```

Knjižnica značk nam ponuja še dve drugi znački, da uporabnikom komponente izpostavimo vsebovane podkomponente: `composite:valueHolder` in `composite:actionSource` [31]. Prva izpostavi izhodne komponente (ang. *output components*), ki imajo nespremenljivo (ang. *non-editable*) vrednost, druga pa komponente, ki prožijo akcijske dogodke (gumbi in povezave). Če želimo uloviti dogodek ob pritisku na gumb za prijavo, spremenimo vmesnik, kot kaže izsek 3.8.

Izsek kode 3.8: Vmesnik komponente z dodano značko `actionSource`

```
<composite:interface>
  <composite:actionSource name="gumbPrijava"
    targets="forma:gumbPrijava"/>
  ...
</composite:interface>
```

Pri uporabi komponente pa dodamo atribut ali značko `actionListener`, ki referencira razred `PrijavaPoslusalec`, v katerem npr. preverimo, ali je uporabnik že registriran. Izsek 3.9 prikazuje registracijo poslušalca `PrijavaPoslusalec` na gumb.

Izsek kode 3.9: Primer registracije poslušalca na gumb

```
<util:prijava ...>
  <f:actionListener for="gumbPrijava"
    type="com.util.PrijavaPoslusalec"/>
</util:prijava>
```

JSF tudi omogoča, da uporabniki komponente lahko spremenijo njen izgled. To nam omogoča uporaba značk JSF facet [18]. V našem primeru želimo uporabniku omogočiti, da lahko vstavi glavo pred formo. Zato ustrezno spremenimo definicijo komponente, kot kaže izsek 3.10.

Izsek kode 3.10: Definicija kompontente z dodanimi značkami JSF facet

```
<composite:interface>
  ...
  <composite:facet name="glava"/>
</composite:interface>
<composite:implementation>
  ...
```

```
   <composite:renderFacet name="glava"/>
   <h:form ...>
      ...
   </h:form>
</composite:implementation>
```

Uporabnik lahko vsebino v glavo komponente doda na način, prikazan v izseku 3.11.

Izsek kode 3.11: Primer dodajanja vsebine v glavo komponente

```
<util:prijava ...>
   <f:facet name="glava" >
      <div>Prosimo, vnesite svoje podatke</div>
   </f:facet>
   ...
</util:prijava>
```

Če pa želimo, da uporabnik doda komponente kot otroke (ang. *children*) v našo komponento, pa spremenimo implementacijo na način, ki je prikazan v izseku 3.12.

Izsek kode 3.12: Definicija komponente z omogočenim dodajanjem otrok

```
<composite:implementation>
   <composite:renderFacet name="glava"/>
   <h:form ...>
      ...
   </h:form>
   <composite:insertChildren/>
</composite:implementation>
```

Uporabnik lahko sedaj v komponento doda poljubno značko. Izsek 3.13 prikazuje primer dodajanja povezave v formo.

Izsek kode 3.13: Primer dodajana povezave v formo

```
<util:prijava...>
   ...
   <f:facet name="glava">...</f:facet>
   <h:link>Povezava</h:link>
</util:prijava>
```

Zelo uporabno bi bilo naši komponenti dodati kodo JavaScript, ki se največkrat uporablja za validacijo na strani odjemalca. To lahko naredimo na zelo preprost

način – v implementaciji naložimo kodo (lahko v posebni datoteki ali pa kar vstavljeno v komponento) in registriramo funkcijo `preveriFormo()`, ki se bo sprožila ob pošiljanju forme [17]. V implementacijo dodamo značke, prikazane v izseku 3.14.

Izsek kode 3.14: Definicija komponente z dodano funkcijo za preverjanje forme

```
<composite:implementation>
   <h:outputScript library="util" name="prijava.js" target="head"/>
   <h:form id="forma"
      onsubmit="return preveriFormo(this, '#{cc.clientId}')">
      ...
   </h:form>
   ...
</composite:implementation>
```

Funkcija `preveriFormo()` dobi dva argumenta: formo, na kateri jo kličemo, in identifikator kompozitne komponente. Funkcija je implementirana v datoteki `prijava.js` na način, kot kaže izsek 3.15.

Izsek kode 3.15: Funkcija `preveriFormo()`

```
function preveriFormo(forma, ccId) {
   var ime = forma[ccId + ':forma:ime'].value;
   var geslo = forma[ccId + ':forma:geslo'].value;
   if (ime == "" || geslo == "") {
      alert("Prosimo, vnesite ime in geslo.");
      return false;
   }
   return true;
}
```

JSF za namen preverjanja veljavnosti vnesenih vrednosti definira komponente za validacijo in pretvorbo, zato jih v praksi raje uporabimo namesto funkcij JavaScript (razen v primeru, da želimo doseči neko dodatno funkcionalnost, ki jih te komponente ne podpirajo). V tem primeru smo uporabili validacijo z uporabo JavaScripta zaradi prikaza načina, kako je mogoče vključiti funkcije JavaScript v kompozitno komponento.

3.3 Izgradnja naprednejših kompozitnih komponent

Včasih hočemo zaradi dodatnega procesiranja vnosov imeti več kontrole nad obnašanjem kompozitne komponente, kot nam jo omogoča sama deklaracija XML. Da lahko dodamo javansko kodo kompozitni komponenti, moramo priskrbeti pomožno komponento (ang. *backing component*). Ta ima naslednje zahteve [29, 31]:

- je podrazred razreda `UIComponent`
- implementira vmesnik `NamingContainer`
- njena lastnost `family` ima vrednost `"javax.faces.NamingContainer"`

Za primer vzemimo kompozitno komponento za vnos datuma rojstva. Vsebuje tri komponente `h:selectOneMenu`, ki sprejmejo podatke tipa Integer (za vnos dneva, meseca in leta). Ker imamo v zrnu `uporabnik` shranjen datum kot objekt Date, bi komponento radi uporabili na način, prikazan v izseku 3.16.

Izsek kode 3.16: Uporaba komponente za vnos datuma rojstva

```
<util:rd vrednost="#{uporabnik.rojstniDan}"/>
```

Definicija komponente je navedena v izseku kode 3.17 (privzamemo, da imamo zrno `datumi`, ki nam napolni komponente `dan`, `mesec` in `leto` z ustreznimi vrednostmi).

Izsek kode 3.17: Definicija komponente za vnos datuma rojstva

```
<composite:interface>
    <composite:attribute name="vrednost" type="java.util.Date"/>
</composite:interface>
<composite:implementation>
   <h:selectOneMenu id="dan" converter="javax.faces.Integer">
     <f:selectItems value="#{datumi.dnevi}"/>
   </h:selectOneMenu>
   <h:selectOneMenu id="mesec" converter="javax.faces.Integer">
     <f:selectItems value="#{datumi.meseci}"/>
   </h:selectOneMenu>
   <h:selectOneMenu id="leto" converter="javax.faces.Integer">
    <f:selectItems value="#{datumi.leta}"/>
   </h:selectOneMenu>
 </composite:implementation>
```

Da bi sestavili javanski objekt Date iz vrednosti treh komponent, potrebujemo pomožno komponento. Ustvarimo jo z novim razredom, ki je ustrezno poimenovan (`rd`) in razširja razred `UIInput`. Ko izrišemo komponento, moramo nastaviti vrednosti podkomponent `dan`, `mesec` in `leto`. To naredimo v razredu pomožne komponente v metodi `encodeBegin()`, ki se izvede v fazi izrisa komponente. V tej metodi pridobimo vrednost kompozitne komponente (torej datum) kot objekt Date, pridobimo komponente `dan`, `mesec` in `leto` kot objekte `UIInput` in preko objekta `GregorianCalendar` vsaki komponenti nastavimo ustrezno vrednost kot objekt Integer in nadaljujemo izris. Ko je forma poslana, moramo rekonstruirati objekt Date iz posameznih izbranih vrednosti. Glede na življenjski cikel JSF bomo to naredili v metodi `getConvertedValue()`, saj so v fazi izvedbe te metode vrednosti podkomponent že pretvorjene v celoštevilski tip. V tej metodi najprej pridobimo vrednosti posameznih podkomponent (`dan`, `mesec` in `leto`) in nato iz njihovih vrednosti s pomočjo objekta `GregorianCalendar` konstruiramo objekt Date. Razred pomožne komponente je prikazan v izseku 3.18.

Izsek kode 3.18: Razred pomožne komponente `rd`

```
public class rd extends UIInput implements NamingContainer {
   public String getFamily() {
      return "javax.faces.NamingContainer";
   }
   public void encodeBegin(FacesContext context) throws IOException {
      Date datum = (Date) getAttributes().get("vrednost");
      Calendar cal = new GregorianCalendar();
      cal.setTime(datum);
      UIInput danKomponenta = (UIInput) findComponent("dan");
      UIInput mesecKomponenta = (UIInput) findComponent("mesec");
      UIInput letoKomponenta = (UIInput) findComponent("leto");
      danKomponenta.setValue(cal.get(Calendar.DATE));
      mesecKomponenta.setValue(cal.get(Calendar.MONTH) + 1);
      letoKomponenta.setValue(cal.get(Calendar.YEAR));
      super.encodeBegin(context);
   }.
   public Object getSubmittedValue() {
      return this;
   }
   protected Object getConvertedValue(FacesContext context,
      Object newSubmittedValue) throws ConverterException {
      UIInput danKomponenta = (UIInput) findComponent("dan");
      UIInput mesecKomponenta = (UIInput) findComponent("mesec");
      UIInput letoKomponenta = (UIInput) findComponent("leto");
```

```
    int dan = (Integer) danKomponenta.getValue();
    int mesec = (Integer) mesecKomponenta.getValue();
    int leto = (Integer) letoKomponenta.getValue();
    return new GregorianCalendar(leto, mesec - 1, dan).getTime();
  }
```

Poleg kompozitnih komponent je od verzije JSF 2.0 naprej novost tudi podpora tehnologiji AJAX. AJAX je tehnologija za posodabljanje vsebine spletne strani brez njenega ponovnega nalaganja (tj. pošiljanja forme in izrisa odgovora) [30]. Stran vsebuje kodo JavaScript, ki komunicira s strežnikom in jo sproti posodablja. Uporabo tehnologije AJAX (in s tem asinhrono posodabljanje vsebine) omogočimo z uporabo značke `h:ajax`, ki jo uporabimo tako kot na običani strani JSF - značko vstavimo v element, ki bo prožil samodejno posodabljanje drugih komponent, določenih z atributom `render`, ob dogodku, določenim z atributom `event`. Možni dogodki, ob katerih lahko prožimo asinhrono posodabljanje, so: `click`, `blur`, `keyup`, `change`, `focus` ipd. (sovpadajo z imeni dogodkov DOM, le da so brez predpone "on") [31]. V izseku 3.19 je podan primer uporabe vgrajene značke AJAX [17].

Izsek kode 3.19: Uporaba tehnologije AJAX v definiciji komponente

```
<composite:implementation>
  ...
  <h:inputText value="#{zrno.lastnost}">
    <f:ajax event="keyup" render="idDrugeKomponente"/>
  </h:inputText>
  ...
</composite:implementation>
```

V tem primeru se ob vsakem pritisku na tipko (ko tipkamo v vnosno polje) ustrezno izriše (posodobi) druga komponenta.

Poglavje 4

Tehnologije na strani odjemalca v kombinaciji z JSF

Realizacijo odzivnih kompozitnih komponent lahko dosežemo z uporabo ene izmed tehnologij (ali njihove kombinacije) na strani odjemalca (ang. *client-side technologies*). Spletne aplikacije kot tehnologijo na strani odjemalca najpogosteje uporabljajo JavaScript. Značilnost teh tehnologij je, da se ne izvajajo na strežniku, temveč pri odjemalcu, natančneje v njegovem brskalniku. Brskalnik lahko zažene interpreter JavaScript in ko se spletna stran, ki vsebuje kodo JavaScript, naloži v brskalnik uporabnika, se interpreter zažene in koda JavaScript izvede [27]. Poleg čistega (ang. *native*) JavaScripta se za programiranje na strani odjemalca v domeni spletnih aplikacij najpogosteje uporabljajo razne knjižnice JavaScript in druge tehnologije, ki olajšajo programiranje, kot so JQuery, AngularJS, Prototype, Bootstrap in podobne [11].

V našem primeru bomo v povezavi z JSF za implementacijo odzivnosti komponent uporabili Bootstrap in JavaScript. Bootstrap bomo uporabili zato, ker vsebuje razrede CSS, ki nam omogočajo načrtovanje tekočega izgleda in ga lahko uporabnik komponente v svojo stran vključi brez večjih sprememb v svoji aplikaciji [17]. JavaScript pa bomo uporabili zato, ker bomo realizirali dokaj preproste funkcije, pri katerih ni potrebna uporaba kakšne naprednejše knjižnice JavaScript, ki bi jo moral uporabnik vključiti v svojo stran. Knjižnico Prototype bomo uporabili samo v primeru, kjer bomo manipulirali z elementi DOM preko objekta `Element`

knjižnice Prototype, saj nam prihrani pisanje specifične kode za vsak brskalnik (ki bi jo morali napisati v čistem JavaScriptu). Za zagotovitev odzivnosti pa bomo uporabili tudi tehnologijo AJAX, ki nam omogoča asinhrono pošiljanje zahtevkov strežniku.

4.1 Programski jezik JavaScript

JavaScript je visokonivojski, dinamični, netipiziran in interpretiran programski jezik, ki je dobro prilagodljiv za objektno orientirano in funkcijsko programiranje [27]. V domeni spletnih aplikacij nam JavaScript omogoča statične strani HTML preurediti v dinamične. Najbolj uporabljana lastnost JavaScripta v povezavi s spletnimi stranmi je interakcija z objektnim modelom strani DOM, s katerim so predstavljeni njeni elementi. Najpogosteje se ta lastnost uporablja za [27]:

- animacijo elementov, saj nam JavaScript omogoča manipulacijo s stilskimi lastnostmi elementov
- interaktivno vsebino in dinamično dodajanje vsebine (dodajanje novih elementov v model DOM)
- validacijo vnesenih vrednosti v forme
- asinhrono posodabljanje strani (v povezavi s tehnologijo AJAX)

Ravno zaradi dejstva, da JavaScript teče lokalno pri uporabnikih, lahko dosežemo zelo veliko odzivnost spletne aplikacije, saj komunikacija s strežnikom ni potrebna. V našem primeru razvoja spletnih aplikacij bomo JavaScript uporabili za validacijo polj komponent, pošiljanje klicev AJAX in manipulacijo z vnesenimi vrednostmi.

4.2 Tehnologija AJAX

AJAX je skupina tehnologij, ki se uporabljajo na strani odjemalca za razvoj asinhronih spletnih aplikacij [28]. Z uporabo tehnologije AJAX lahko aplikacije pošiljajo in pridobivajo podatke strežnika asinhrono v ozadju brez motenja obnašanja obstoječe strani. Kratica AJAX pomeni asinhroni JavaScript in XML,

saj največkrat uporablja ti dve tehnologiji. Poleg tehnologije XML se za prenos podatkov lahko uporablja tudi JSON. AJAX vključuje naslednje tehnologije [28]:

- HTML in CSS za standarden prikaz strani
- model DOM za prikaz in interakcijo z elementi strani
- XML za izmenjavo podatkov
- JavaScript za povezovanje zgoraj naštetega

JavaScript se pri tehnologiji AJAX uporablja za manipulacijo z elementi DOM (pridobivanje in nastavljanje njihovih vrednosti) in interakcijo z objektom `XMLHttpRequest`, s katerim ponujata metodo za asinhrono izmenjavo podatkov med strežnikom in brskalnikom. AJAX eliminira t.i. start-stop naravo interakcije s spletnimi stranmi z dodajanjem vmesnega sloja – t.i. pogona AJAX (ang. *AJAX engine*) med uporabnikom in strežnikom. Namesto nalaganja spletne strani na začetku seje brskalnik zažene pogon AJAX, ki skrbi za izris strani in komunikacijo s strežnikom [28, 4]. Vsako uporabnikovo dejanje, ki bi generiralo zahtevo HTTP, sproži klic pogonu preko JavaScripta. Vsak odgovor na uporabnikovo dejanje, ki ne potrebuje poti do strežnika, prevzame pogon. Če potrebuje podatke s strežnika, da bi odgovoril uporabniku, sproži asinhrono zahtevo z uporabo tehnologije XML strežniku, ne da bi se uporabnik tega zavedal. V našem primeru bomo AJAX uporabili pri primeru komponente, ki zahteva takojšen odziv in pridobiva podatke s strežnika.

4.3 Ogrodje Bootstrap

Bootstrap je odprtokodno ogrodje HTML, CSS in JavaScript (ang. *framework*) za razvoj odzivnih spletnih aplikacij [2]. Vključuje predloge HTML in CSS izgleda spletnih strani in njihovih elementov (form, gumbov itd.) kot tudi JavaScript razširitve. Trenutno je v uporabi verzija 3.2.0, ki jo bomo uporabljali pri izgradnji odzivnih komponent. Začenši z verzijo 2.0 Bootstrap podpira odziven izgled spletnih strani, kar pomeni, da se izgled spletnih strani samodejno prilagaja glede na uporabljano napravo oz. velikost zaslona. To dosežemo s tem, da elemente HTML

opremimo z ustreznimi razredi CSS, ki jih vsebuje Bootstrap. Bootstrap je modularen in je sestavljen predvsem iz stilskih predlog LESS (ang. *LESS stylesheets*) [2]. LESS je predprocesor CSS, ki razširja jezik CSS in nam omogoča izgradnjo dinamičnih predlog z dodajanjem spremeljivk, funkcij, vejitev in različnih drugih tehnik [22].

Ogrodje Bootstrap vsebuje razne razrede CSS, ki nam ponujajo različne možnosti spremembe izgleda komponentam HTML in ostalih sestavnih delov spletnih strani (ozadje, pisava, postavitev vsebine itd.). V našem primeru bomo komponente opremili z razredi CSS, ki nam bodo omogočili postavitev besedila na sredino vsebovalnika (razred `.text-center`) in lepši stil ter izgled komponent (npr. razred `.btn-primary` za gumb). Uporabili bomo tudi razrede CSS za forme, ki nam bodo omogočali lepši stil njenih komponent (razred `.form-control` omogoča vnosnim poljem, da zasedejo celotno širino svojega odseka, v razred `.form-group` pa bomo ovili kontrole z njihovimi labelami za optimalno postavitev). Ker bomo želeli imeti naše elemente forme horizontalno poravnane, bomo uporabili razred `.form-horizontal` [2].

Bootstrap se v zadnjih verzijah vedno bolj posveča načrtovanju izgleda, ki je že od začetka prilagojen za mobilne naprave (ang. *mobile-first design*). Pravilna uporaba ogrodja Boostrap za dosego odzivnosti zahteva vsebovalnik, ki ovije elemente strani in obdaja mrežni sistem (ang. *grid system*). Če želimo spremeniti odsek strani v odzivni vsebovalnik, ga opremimo z razredom `.container` oz. `.container-fluid` [3]. Bootstrap uporablja odziven mrežni sistem, ki je sestavljen iz do dvanajstih stolpcev (ob povečevanju zaslona oz. odseka, v katerem se nahaja, se povečuje tudi njihovo število). Mrežni sistem se uporablja za izgradnjo načrta postavitve strani skozi zaporedje vrstic in stolpcev, ki vsebujejo vsebino strani. Princip delovanja sistema je naslednji [2, 3]:

- vrstice (ang. *rows*) morajo biti postavljene v vsebovalnik, opremljen z razredom `.container-fluid`
- z vrsticami ustvarimo horizontalne skupine stolpcev
- vsebina se nahaja znotraj stolpcev, ki so neposredni otroci vrstic
- za hitro izgradnjo načrta postavitve so na voljo razredi kot `.row` in `.col`

- stolpce ustvarimo z navedbo števila (do dvanajst) možnih stolpcev, čez katere se bo naš stolpec raztezal. Če hočemo ustvariti tri enake stolpce, bodo uporabljali razred `.col-xs-4`
- če v vrsto postavimo več kot dvanajst stolpcev, se ti prenesejo v naslednjo vrsto
- uporabljeni razredi veljajo za naprave z zaslonom, ki je enak ali večji od navedenega (drugi del imena razreda stolpca)

Pri izgradnji odzivnih kompozitnih komponent bomo, kjer bo primerno, uporabili mrežni sistem ogrodja Boostrap za realizacijo t.i. tekočega izgleda (ang. *fluid design*), pri katerem se komponente razporedijo tako, da zasedejo celotno širino odseka, ki je bil dodeljen naši komponenti.

Poglavje 5

Primeri uporabnih kompozitnih komponent

V tem poglavju bomo prikazali izdelavo štirih zanimivih in uporabnih kompozitnih komponent, ki bodo v praksi prikazale vse pristope njihovega razvoja, ki so bili omenjeni v prejšnjih poglavjih. Pri vsaki komponenti bomo najprej podali primer njene uporabe in odgovor na vprašanje, zakaj je komponenta uporabna. Zraven bo podan opis razvoja posamezne komponente, slika njenega prikaza (kjer bo smiselno tudi slika prikaza v mobilnem načinu) in ključni izseki programske kode. Programsko okolje, ki ga bomo uporabljali ob njihovi izdelavi, je Eclipse [14], uporabljali bomo programsko platformo Java EE 7 [10] in implementacijo JSF 2.2 Mojarra. Aplikacijo JSF bomo testirali na strežniku Glassfish v4.0 [7].

5.1 Kompozitna komponenta za vnos datuma

Vnos datuma je pogosto opravilo uporabnikov spletnih strani. Težava pa pogosto nastane v tem, da mora uporabnik vnesti datum v pravilnem formatu, da ga naša aplikacija lahko sprejme oz. mora razvijalec aplikacije razviti logiko, ki bo znala prebrati uporabnikov vnos in ga ustrezno formatirati. Te težave se lahko rešimo z uporabo koledarjev, ki uporabljajo JavaScript, ali pa s tremi vnosnimi polji za vnos dneva, meseca in leta. Slednje je ponavadi nepriljubljena izbira, saj zahteva daljši čas in manj prijazno implementacijo. Zato bi bilo smiselno izgraditi kompozitno

komponento za vnos datuma, ki bi vsebovala meni za izbiro dneva, meseca in leta. Komponenta se nanaša na komponento, ki je bila v poglavju 3.3 uporabljena za opis izgradnje naprednejših komponent s pomočjo t.i. pomožnega zrna. Kot bo razvidno iz implementacije, gre za zelo preprost način uporabe kompozitnih komponent s pomožnim zrnom v praksi. Komponento želimo uporabiti na način, prikazan v odseku 5.1.

Izsek kode 5.1: Uporaba komponente za vnos datuma

```
<util:datum id="datum" vrednost="#{uporabnik.izbranDatum}"/>
```

Uporabniku komponente hočemo omogočiti čim bolj preprosto uporabo komponente, zato želimo, da nam komponenta vrne vrednost izbranega datuma kot objekt `Java.util.Date`, saj tako uporabnik ne bo imel težav s pretvarjanjem formatov zapisa.

Najprej definiramo komponento v datoteki `datum.xhtml` (izsek 5.2), kjer ji definiramo atribut `vrednost` ter jo implementiramo kot tri menije za izbiro `h:selectOne` z imeni `dan`, `mesec` in `leto`, ki bodo vsebovali podatek o izbranem dnevu, mesecu in letu. Za napolnitev posameznih menijev z dnevi, meseci in leti uporabimo pomožno zrno `Datumi.java`. Pri tej komponenti zaradi njene preprostosti odzivnost ne pride do izraza, vendar bomo izgradnjo odzivnih komponent predstavili že pri naslednjem primeru komponente, t.i. tekoči izgled pa pri tretji komponenti (registracija). V tem primeru jo ovijemo le v element `h:panelGrid` z razredom CSS `.container-fluid` – tu seveda uporabimo knjižnico Bootstrap.

Izsek kode 5.2: Datoteka `datum.xhtml` (definicija komponente)

```
<composite:interface>
  <composite:attribute name="vrednost" type="java.util.Date" />
</composite:interface>
<composite:implementation>
  <h:panelGroup layout="block" class="container-fluid">
        <h:selectOneMenu id="dan">
          <f:selectItems value="#{datumi.dnevi}"/>
        </h:selectOneMenu>
        <h:selectOneMenu id="mesec">
          <f:selectItems value="#{datumi.meseci}"/>
        </h:selectOneMenu>
        <h:selectOneMenu id="leto">
          <f:selectItems value="#{datumi.leta}"/>
        </h:selectOneMenu>
```

```
    </h:panelGroup>
  </composite:implementation>
```

Ker pa dosedanja implementacija v datoteki XHTML še ni konfigurirana, da nam komponenta vrača objekt Date, potrebujemo pomožno komponento. To ustvarimo na način, ki je bil opisan v poglavju 3.3 v datoteki `datum.java` (razred je poimenovan z malo začetnico zaradi upoštevanja pravila konvencije pred konfiguracijo (ang. *convention over configuration*), saj je komponenta definirana v imenskem prostoru `util.datum`). Skelet razreda je prikazan v izseku 5.3.

Izsek kode 5.3: Skelet razreda pomožne komponente

```
public class datum extends UIInput implements NamingContainer {
  public String getFamily() {
    return "javax.faces.NamingContainer";
  }
  public void encodeBegin(FacesContext context) throws IOException
  {
    ...
  }
  public Object getSubmittedValue() {
    return this;
  }
  protected Object getConvertedValue(FacesContext context,
        Object newSubmittedValue) throws ConverterException {
       ...
  }
}
```

Metoda `encodeBegin()` se izvede v fazi izrisa komponente, zato moramo v njej nastaviti vrednosti podkomponent, ki jih dobimo iz vrednosti atributa `vrednost` (to vrednost dobimo iz uporabnikovega zrna). V metodi najprej pridobimo vrednost atributa `vrednost` kot objekt Date. Nato pridobimo vsako podkomponento kot primerek objekta `UIInput` in ji s pomočjo objekta `GregorianCalendar` nastavimo vrednosti za dan, mesec in leto, ki smo jih dobili iz podanega datuma ter nadaljujemo z izrisom. Metoda `encodeBegin()` je prikazana v izseku 5.4.

Izsek kode 5.4: Metoda `encodeBegin()`

```
public void encodeBegin(FacesContext context) throws IOException
{
  Date datum = (Date)getAttributes().get("vrednost");
  // ustvarimo pomozen objekt GregorianCalendar za razclenitev datuma
```

```
  Calendar calendar = new GregorianCalendar();
  calendar.setTime(datum);

  // pridobitev komponent in nastavitev njihovih vrednosti
  UIInput d = (UIInput)findComponent("dan");
  UIInput m = (UIInput)findComponent("mesec");
  UIInput l = (UIInput)findComponent("leto");
  d.setValue(calendar.get(Calendar.DATE));
  m.setValue(calendar.get(Calendar.MONTH) + 1);
  l.setValue(calendar.get(Calendar.YEAR));
  super.encodeBegin(context);
}
```

Ko je forma poslana, se zažene življenjski cikel JSF in je potrebno konstruirati objekt Date iz posameznih vnosnih polj, saj se bo ta vrednost nastavila v uporabnikovem zrnu. Pretvorbo vrednosti v objekt Date bomo izvedli v metodi `getConvertedValue()`, saj se v tej fazi (tretja faza življenjskega cikla) izvajajo pretvorbe. V našem primeru moramo seveda iz komponent `dan`, `mesec` in `leto` dobiti objekte Date. V tej metodi najprej pridobimo podkomponente `dan`, `mesec` in `leto` kot objekt `UIInput` in nato njihove poslane vrednosti, ki so tipa String. Nato je potrebno preveriti, ali je vneseni datum pravilen. Najprej iz posameznih delov sestavimo niz, ki predstavlja vneseni datum in ga pretvorimo v objekt Date s pomočjo objekta `SimpleDateFormat`. Slednjega nastavimo tako, da ne bo poskušal popraviti napačnega datuma, ampak bo v tem primeru sprožil izjemo `ParseException` (ukaz `f.setLenient(false)`). V primeru vnosa veljavnega datuma vrnemo ustrezen objekt Date. Če pa datum ni veljaven, sprožimo napako pri pretvorbi (`ConverterException`) z ustreznim sporočilom (objekt `FacesMessage` z ustrezno resnostjo napake). Implementacijo metode `getConvertedValue()` prikazuje izsek 5.5.

Izsek kode 5.5: Metoda `getConvertedValue()`

```
protected Object getConvertedValue(FacesContext context,
    Object newSubmittedValue) throws ConverterException {

  UIInput dKomp = (UIInput)findComponent("dan");
  UIInput mKomp = (UIInput)findComponent("mesec");
  UIInput lKomp = (UIInput)findComponent("leto");
  String d = (String) danKomponenta.getSubmittedValue();
  String m = (String) mesecKomponenta.getSubmittedValue();
  String l = (String) letoKomponenta.getSubmittedValue();
```

```
  // preverimo veljavnost datuma s tem, da ga skusamo pretvoriti v objekt Date
  String izbranDatum = l + "-" + m + "-" + d;
  DateFormat f = new SimpleDateFormat("yyyy-MM-dd");
  f.setLenient(false);
  try {
    Date dat = f.parse(izbranDatum);
    return dat;
  } catch (ParseException ex) {
    FacesMessage message = new FacesMessage("Napacen datum!");
    message.setSeverity(FacesMessage.SEVERITY_ERROR);
    throw new ConverterException(message);

  }
}
```

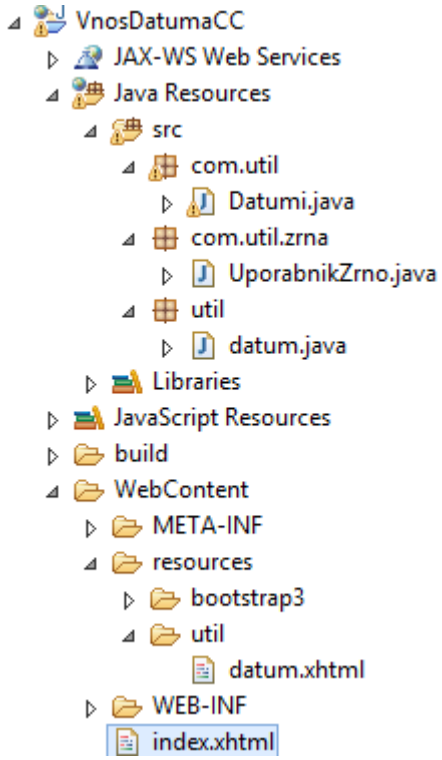
Na koncu ustvarimo stran JSF z dodanim gumbom in naslovom, kjer komponento uporabimo (izsek 5.6).

Izsek kode 5.6: Datoteka `index.xhtml` (stran JSF z uporabo komponente)

```
<h:body>
   <h:form>
      <h3>Vnesite datum.</h3>
      <util:datum id="datum" value="#{uporabnik.izbranDatum}"/>
      <br/><h:message for="datum" />
      <p><h:commandButton value="Naprej" action=" "/></p>
   </h:form>
</h:body>
```

Kot rezultat dobimo komponento na sliki 5.1. Uporabnik lahko s klikom na posamezni meni izbere dan, mesec in leto, s klikom na gumb 'Naprej' pa sproži pošiljanje forme. Celotna struktura projekta se nahaja na sliki 5.2.

Slika 5.1: Prikaz komponente za vnos datuma



Slika 5.2: Struktura projekta s kompozitno komponento za vnos datuma

5.2 Kompozitna komponenta spinner

JSF vsebuje svoj nabor značk, a njegova osnovna verzija sama po sebi še ne podpira vseh elementov HTML5 [8]. Boljšo podporo elementom HTML5 omogočajo njegove razširitve oz. ogrodja (npr. PrimeFaces), neposredno podporo elementom HTML5 pa je moč doseči tudi v JSF 2.2, a je zato potrebnih nekaj popravkov v spletni aplikaciji. Med elementi HTML5 je eden bolj uporabnih element spinner, ki se uporablja za vnos števil, kjer imamo končno množico števil, ki se med seboj razlikujejo za določen korak. Zato bi bilo tudi v JSF uporabno razviti kompozitno komponento, ki bi imela isto funkcionalnost kot element spinner in za njeno uporabo avtorju spletne aplikacije ne bi bilo potrebno popravljati obstoječe aplikacije.

Realizacija je preprosta: ustvarimo kompozitno komponento z atributi `vrednost`, `korak`, `min` in `max` (meje vrednosti, ki jih sprejmemo). Atribut `vrednost` je zahtevan in ima privzeto vrednost 0. Komponento implementiramo s pomočjo

vnosnega polja in gumbov za povečanje oziroma zmanjšanje vrednosti (izsek 5.7).

Izsek kode 5.7: Datoteka `spinner.xhtml` (definicija komponente)

```
<composite:interface>
   <composite:attribute name="vrednost" required="true" default="0"/>
   <composite:attribute name="korak" required="false"/>
   <composite:attribute name="max" required="false" />
   <composite:attribute name="min" required="false" />
</composite:interface>
<composite:implementation>
   <h:panelGroup layout="block" class="container-fluid">
      <h:inputText id="nID" value="#{cc.attrs.vrednost}"
      <h:commandButton id="lID" value="-" />
      <h:commandButton id="rID" value="+" />
   </h:panelGroup>
</composite:implementation>
```

Ker pa hočemo imeti odzivno komponento in nočemo ob vsakem spreminjanju vrednosti poslati vrednosti na strežnik, na strani odjemalca registriramo rokovalnik dogodka – torej funkcijo JavaScript z imenom `spremeniVrednost()`, ki se bo sprožila ob kliku na gumb za povečanje oz. zmanjšanje vrednosti. Kot argumente bo sprejela število -1 ali 1 (zmanjšanje ali večanje vrednosti, odvisno od kliknjenega gumba), korak, meje in identifikator komponente. V komponento tudi vključimo datoteko JavaScript, v kateri bo funkcija implementirana. Spremenjena definicija komponente je prikazana v odseku 5.8.

Izsek kode 5.8: Spremenjena definicija komponente

```
<composite:implementation>
   <h:outputScript library="util" name="spinner.js" />
   <h:panelGroup layout="block" class="container-fluid">
      ...
      <h:commandButton id="lID" value="-" onclick="return
          spremeniVrednost(-1, '#{cc.attrs.korak}', '#{cc.attrs.max}',
          '#{cc.attrs.min}', '#{cc.clientId}')"
      />
      <h:commandButton id="rID" value="+" onclick="return
        spremeniVrednost(1, '#{cc.attrs.korak}', '#{cc.attrs.max}',
        '#{cc.attrs.min}', '#{cc.clientId}')"
      />
   </h:panelGroup>
</composite:implementation>
```

Funkcijo `spremeniVrednost()` zapišemo v svojo datoteko `spinner.js`. Funkcija bo poskrbela za povečanje oz. zmanjšanje vnesene vrednosti (odvisno od gumba, na katerega uporabnik klikne) za določen korak in to, da ne moremo preseči mej, v kolikor jih je uporabnik komponente določil. V funkciji najprej preverimo, če so bile vnesene vrednosti atributov `korak`, `max` in `min` in jih spremenimo v številski tip. V primeru, da korak ni bil vnesen, ga privzeto nastavimo na 1. Nato pridobimo vnosno polje glede na njegov identifikator (sestavljen je iz identifikatorja komponente, dvopičja in njegovega identifikatorja - 'nID') in njegovo vrednost povečamo za produkt koraka in spremembe vrednosti (1 ali -1). Nato pa še v primeru, da so bile vnesene vrednosti za meje, preverimo, da jih vrednost komponente ne presega. Implementacija funkcije `spremeniVrednost()` je prikazana v izseku 5.9.

Izsek kode 5.9: Datoteka `spinner.js` (funkcija `spremeniVrednost()`)

```
function spremeniVrednost(x, k, ma, mi, cid) {
	var niMax = false;
	var niMin = false;
	var korak = Number(k);
	if(isNaN(korak) || korak == 0) korak = 1;
	var max = Number(ma);
	if(isNaN(max)) niMax = true;
	var min = Number(mi);
	if(isNaN(min)) niMin = true;
	var spinner = document.getElementById(cid + ":" + "nID");
	spinner.value = Number(spinner.value) + (x * korak);
	if(!niMax) {
		if(Number(spinner.value) >= max){
			spinner.value = max;
		}
	}
	if(!niMin){
		if(Number(spinner.value) < min){
			spinner.value = min;
		}
	}
	return false;
}
```

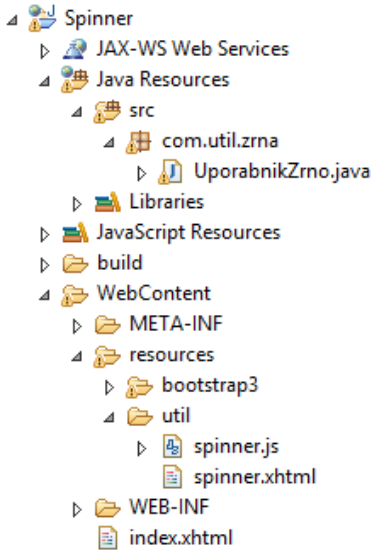
Nato ustvarimo še stran JSF, v kateri komponento uporabimo (izsek 5.10).

Izsek kode 5.10: Datoteka `index.xhtml` (uporaba komponente v strani JSF)

```
<h:body>
  <h:panelGroup layout="block" class="container-fluid">
    <h:panelGroup layout="block" class="row">
      <h:outputLabel >Izberi vrednost:</h:outputLabel>
      <util:spinner vrednost="#{uporabnik.vrednost}" korak="1" max="8" min="0"/>
    </h:panelGroup>
  </h:panelGroup>
</h:body>
```

Kot rezultat dobimo komponento na sliki 5.3. V našem primeru uporabe komponente v izseku 5.10 bo uporabnik lahko povečeval in zmanjševal vrednost komponente za korak 1 med vrednostmi 0 in 8. Celotno strukutro projekta prikazuje slika 5.4.

Slika 5.3: Kompozitna komponenta spinner



Slika 5.4: Struktura projekta kompozitna komponenta spinner

5.3 Kompozitna komponenta za registracijo uporabnikov

Registracija je prisotna skoraj pri vseh spletnih aplikacijah, kjer hočemo imeti dinamično vsebino, ki jo soustvarjajo uporabniki, ali tam, kjer jim nudimo naše storitve. Zato je pri skoraj vsaki izdelavi spletne aplikacije potrebna tudi izdelava obrazca za vnos uporabnikovih podatkov pri registraciji. To je za razvijalce pogosto obremenjujoče opravilo, saj je vedno potrebno ustvariti formo z vnosnimi polji in pri tem poskrbeti še za njihovo obliko oz. postavitev. Zato bi bilo smiselno zgraditi komponento, ki bi sama izrisala vedno prisotna polja za vnos uporabnikovih podatkov in bi jih lahko uporabnik (tj. razvijalec spletne aplikacije) le vključil v svojo aplikacijo in ustrezno povezal s svojim zrnom. Obrazec za prijavo je bil predstavljen že v poglavju 3.2, zato bomo tokrat prikazali izdelavo obrazca za registracijo. Komponento želimo uporabiti na način, ki ga prikazuje izsek 5.11.

Izsek kode 5.11: Uporaba kompozitne komponente za registracijo

```
<util:registracija
   email="#{uporabnik.email}"
   geslo="#{uporabnik.geslo}"
   ime="#{uporabnik.ime}"
   priimek="#{uporanik.priimek}"
   gumbRegistracijaOpis="Registriraj se"
   gumbRegistracijaAkcija="#{uporabnik.registracija}" >
   <!-- validatorji, faceti itd. -->
</util:registracija>
```

V začetni verziji komponente bomo njenemu uporabniku ponudili polja, ki so ob registraciji uporabnika vedno prisotna, tj. vnos elektronskega naslova, gesla, imena in priimka. Omogočili mu bomo tudi vpis lastnega besedila na gumb ter registracijo lastnega dogodka ob kliku na gumb. Najprej razvijemo osnovno komponento po standardnem postopku. Definicija kompozitne komponente se bo nahajala v datoteki `registracija.xhtml` v mapi *resources/util*. V njej definiramo atribute komponente `email`, `geslo`, `ime`, `priimek`, `gumbRegistracijaOpis` (napis na gumbu) in `gumbRegistracijaAkcija` (metoda, ki se bo prožila ob kliku na gumb), s katerimi bo uporabnik komponento povezal s svojim zrnom, in implementacijo, ki je seveda forma z vnosnimi polji in pripradajočimi labelami za vnos

el. naslova, vnos gesla, ponovitev gesla, vnos imena in vnos priimka ter gumbom za potrditev oz. pošiljanje vnesenih podatkov na strežnik. Polja za vnos elektronskega naslova, gesla in ponovitev gesla nastavimo kot zahtevana. Izsek 5.12 prikazuje začetno verzijo definicije komponente.

Izsek kode 5.12: Datoteka `registracija.xhtml` (definicija komponente)

```
<composite:interface>
  <composite:attribute name="email" />
  <composite:attribute name="geslo" />
  <composite:attribute name="ime" />
  <composite:attribute name="priimek" />
  <composite:attribute name="gumbRegistracijaOpis" />
  <composite:attribute name="gumbRegistracijaAkcija"
    method-signature="java.lang.String action()" />
</composite:interface>
<composite:implementation>
  <h:form id="forma" >
    <h:outputLabel id="email_labela" for="email"> E-mail: </h:outputLabel>
    <h:inputText id="email" value="#{cc.attrs.email}" required="true" />
    <h:outputLabel id="geslo1_labela" for="geslo1"> Geslo: </h:outputLabel>
    <h:inputSecret id="geslo1" value="#{cc.attrs.geslo}" required="true" />
    <h:message for="geslo1"/>
    <h:outputLabel id="geslo2_labela" for="geslo2"> Ponovite geslo: </h:outputLabel>
    <h:inputSecret id="geslo2" value="#{cc.attrs.geslo}" required="true" />
    <h:outputLabel id="ime_labela" > Ime: </h:outputLabel>
    <h:inputText id="ime" value="#{cc.attrs.ime}" />
    <h:outputLabel id="priimek_labela" for="priimek"> Priimek: </h:outputLabel>
    <h:inputText id="priimek" value="#{cc.attrs.priimek}" />
    <h:commandButton id="gumbRegistracija"
       action="#{cc.attrs.registerButtonAction}"
       value="#{cc.attrs.registerButtonText}" />
  </h:form>
</composite:implementation>
```

Trenutno izdelana komponenta sicer njenemu uporabniku prihrani nekaj dela, vendar je zelo neprilagodljiva in neprijazna za uporabo. Najprej bi bilo smiselno omogočiti dodajanje validatorjev na posamezna vnosna polja, saj bo njen uporabnik verjetno želel preveriti pravilnost vnesenega el. naslova, dolžino gesla, ujemanje gesel itd. Zato v vmesnik komponente vključimo polja za izpostavljanje smiselnih podkomponent, v našem primeru posebej izpostavimo vnosno polje za vnos el. naslova (uporabno za preverjanje pravilnosti naslova), posebej polje za vnos gesla (uporabno za preverjanje dolžine gesla), skupaj polji za vnos in ponovitev gesla

(preverjanje enakosti) ter nato še vsa polja skupaj (izsek 5.13).

Izsek kode 5.13: Izpostavljene podkomponente v njeni definiciji

```
<composite:interface>
  <composite:editableValueHolder name="emailVnos"
    targets="forma:email"/>
  <composite:editableValueHolder name="gesloVnos"
    targets="forma:geslo1"/>
  <composite:editableValueHolder name="gesloVnosi"
    targets="forma:geslo1 forma:geslo2"/>
  <composite:editableValueHolder name="vnosi"
    targets="forma:email forma:geslo1 forma:geslo2
             forma:ime forma:priimek"/>
  ...
</composite:interface>
```

Tako bo lahko uporabnik komponente ob njeni uporabi registriral razne validatorje na polja forme, ki smo jih izpostavili z zgornjimi značkami.

Predvidevamo lahko, da bo uporabnik komponente želel preveriti, ali je kakšen uporabnik že registriran pod vnesenim el. naslovom. Zato bi bilo tudi smiselno izpostaviti vire dogodkov (ang. *action sources*) naše komponente, na katere bo lahko uporabnik komponente registriral poslušalce. V našem primeru izpostavimo gumb za registracijo kot vir dogodka (izsek 5.14).

Izsek kode 5.14: Izpostavljanje vira dogodka v definiciji komponente

```
<composite:interface>
  ...
  <composite:actionSource name="gumbRegistracija" targets="forma:gumbRegistracija"/>
</composite:interface>
```

Smiselno bi bilo tudi uporabniku omogočiti spreminjanje komponente z dodajanjem elementov JSF facet, v našem primeru za vnos glave forme in izpisa napak na koncu forme. V vmesniku komponente definiramo dve znački `<composite:facet>` za glavo in izpis napak ter uporabimo značko `<composite:renderFacet>` na mestu v implementaciji, kjer bo vsebina glave oz. napak izpisana (v našem primeru na vrhu forme in za njo). Ker bo uporabnik komponente verjetno želel od uporabnika aplikacije tudi kakšno drugo informacijo (npr. vnos uporabniškega imena, telefonske številke, naslova itd.), mu bomo omogočili dodajanje novih vnosnih polj oz. elementov kot otrok v formo. V implementaciji komponente na

mestu med zadnjim vnosnim poljem ter potrditvenim gumbom dodamo element `<composite:insertChildren/>`, s čimer na tem mestu uporabniku komponente omogočimo dodajanje otrok v formo. Spremenjeno definicijo komponente prikazuje izsek 5.15.

Izsek kode 5.15: Definicija komponente z omogočenim dodajanjem elementov facet in otrok

```
<composite:interface>
  ...
  <composite:facet name="glava"/>
  <composite:facet name="napaka"/>
</composite:interface>
<composite:implementation>
  <h:form id="forma" >
    <composite:renderFacet name="glava" />
    ...
    <h:inputText id="priimek" ... />
    <composite:insertChildren />
    <h:commandButton ... />
  </h:form>
  <composite:renderFacet name="napaka"/>
</composite:implementation>
```

Ker je pri razvoju form za registracijo splošna praksa, da uporabnik dvakrat vnese svoje geslo za potrditev, bi bilo smiselno, da bi komponenta že sama preverila enakost gesel. Za ta namen je najbolje uporabiti validacijo z uporabo JavaScripta, ki bo vključen že v samo komponento. V isti mapi kot se nahaja definicija komponente, ustvarimo datoteko JavaScript s funkcijo `preveriFormo()`, ki bo preverila enakost gesel. Funkcija bo kot argumente sprejela formo in identifikator kompozitne komponente. V funkciji najprej pridobimo obe vnosni polji iz forme glede na njihov identifikator (sestavljen iz identifikatorja kompozitne komponente, dvopičja, identifikatorja forme, kateremu sledi še eno dvopičje in identifikatorja vnosnega polja) in njihovi vrednosti. Nato preverimo, da sta obe vnosni polji izpolnjeni ter da se vnosa ujemata. Samo v tem primeru dovolimo pošiljanje forme. Funkcijo `preveriFormo()` prikazuje izsek 5.16.

Izsek kode 5.16: Datoteka `registracija.js` (validacija gesla v komponenti)

```
function preveriFormo(form, ccId) {
   var geslo1 = form[ccId + ':forma:geslo1'].value;
```

```
    var geslo2 = form[ccId + ':forma:geslo2'].value;

    if (geslo1 == "" || geslo2 == "") {
        alert("Prosimo, izpolnite obe polji za geslo.");
        return false;
    }
    if(geslo1 !== geslo2) {
            alert("Gesli se ne ujemata!");
            return false;
    }
    return true;
}
```

Nato moramo še datoteko `registracija.js` vključiti v implementacijo komponente ter registrirati funkcijo `preveriFormo()` na dogodek, ki se sproži ob pošiljanju forme ('submit') (izsek 5.17).

Izsek kode 5.17: Registracija funkcije `preveriFormo()`

```
<composite:implementation>
  <h:outputScript library="util" name="registracija.js" />
  <h:form id="forma" onsubmit="return preveriFormo(this, '#{cc.clientId}')" >
    ...
  </h:form>
  ...
</composite:implementation>
```

Ker razvijamo odzivne kompozitne komponente, moramo poskrbeti tudi za tekoči izgled, pri katerem z uporabo knjižnice Bootstrap omogočimo komponenti, da se odziva na velikost zaslona uporabnikove naprave. Omogočili bomo raztezanje komponente v oknu oz. odseku, ki ga bo določil uporabnik komponente. V tem odseku bo komponenta zasedla celoten prostor po širini, pri čemer se bodo vnosna polja in labele ustrezno razporedili, tako da bodo na sredini odseka in na način, da uporabniku na napravah z manjšim zaslonom (telefoni, tablični računalniki itd.) ne bo potrebno uporabljati drsnika za premikanje po strani, kar je zelo neprijetna uporabniška izkušnja. To storimo z opremljanjem komponente z elementi `<h:panelGrid layout='block' >` in ustreznim razredom iz knjižnice Bootstrap, ki bo (ob pravilni konfiguraciji elementov z razredi) poskrbela za odzivnost komponente. Formi nastavimo razred `.class-horizontal`, saj želimo, da bo horizontalno poravnana. Nato ustvarimo odsek (element `panelGrid`) z razredom `.row`, saj hočemo komponente razporediti v odzivno vrsto (uporaba mrežnega sistema

Bootstrap). Nato ustvarimo odsek z razredoma `.col-sm-6` in `.col-lg-4`, v katerega damo nov odsek, ki vsebuje vnosni element s pripadajočo labelo (ta odsek ima zato razred `.form-group`). Uporaba zgornjih razredov pomeni, da bosta vnosno polje in labela na manjših napravah zasedla polovico vrstice, na večjih pa tretjino. Znotraj te vrstice bo labela na srednje velikih napravah zasedla eno tretjino prostora (razred `.col-md-4`), polje pa dve (razred `.col-md-8`). Na napravah drugih velikosti se bo labela nahajala takoj nad poljem. Nato postavimo gumb v svojo vrstico (na sredino) in ga opremimo z razredoma `.btn .btn-primary`, s katerima mu polepšamo izgled in ga poudarimo. V izseku 5.18 je podana spremenjena definicija komponente, ki uporablja tekoči izgled (podan je primer le za eno vnosno polje, saj se za vsako polje opremljanje z odseki in razredi ponovi).

Izsek kode 5.18: Definicija komponente z uporabo tekočega izgleda

```
<composite:implementation>
  <h:form id="forma" class="form-horizontal" role="form"
    onsubmit="return preveriFormo(this, '#{cc.clientId}')" >
    <h:panelGroup layout="block" class="row">
      <h:panelGroup layout="block" class="col-sm-6 col-lg-4">
        <h:panelGroup layout="block" class="form-group required">
          <h:outputLabel id="email_labela" for="email"
            class="col-md-4 control-label">E-mail: </h:outputLabel>
          <h:panelGroup layout="block" class="col-md-8">
            <h:inputText id="email" value="#{cc.attrs.email}" required="true"
              class="form-control"/>
          </h:panelGroup>
          <h:message for="email"/>
        </h:panelGroup>
      </h:panelGroup>
      ...
    </h:panelGroup> <!-- vrstica -->
    <composite:insertChildren/>
    <h:panelGroup layout="block" class="row" style="text-align:center">
       <h:commandButton id="gumbRegistracija"
         action="#{cc.attrs.registerButtonAction}"
         value="#{cc.attrs.registerButtonText}" class="btn btn-primary"
        />
    </h:panelGroup>
  </h:form>
  <composite:renderFacet name="napaka"/>
</composite:implementation>
```

Za uporabo knjižnice Bootstrap (in s tem odzivnosti komponente) mora uporabnik komponente v stran, v kateri bo uporabljal komponento, le vključiti datoteko CSS `bootstrap.css` ter našo komponento oviti v element `h:panelGrid` z ustrezno velikostjo in razredom `.container-fluid`. Nato ustvarimo stran JSF, kjer uporabimo komponento (v našem primeru je območje za komponento cel zaslon) v odzivnem odseku, ki je sredinsko poravnan (razred `.container-fluid`). V stran še vključimo knjižnico Boostrap. V našem primeru uporabe komponente na strani je prikazan še primer dodajanja validatorjev na komponento. Z značko `f:validateLength` preverimo dolžino gesel, z značko `f:validateRegex` pa pravilnost el. naslova. Prikazan je tudi primer dodajanja lastnega validatorja (značka `f:validator`) na polji z gesli. Prikazana je tudi registracija poslušalca na klik gumba (značka `f:actionListener`) in elementa facet za izpis morebitnih sporočil pri pošiljanju forme. Uporaba komponente na strani je prikazana v izseku 5.19.

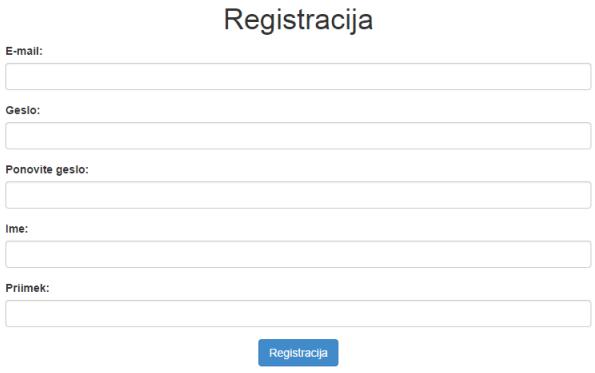
Izsek kode 5.19: Datoteka `index.xhtml` (uporaba komponente)

```
<h:head>
  <h:outputStylesheet library="bootstrap3" name="bootstrap.css" />
</h:head>
<h:body>
  <h:panelGroup layout="block" class="container-fluid">
    <h:panelGroup layout="block" class="text-center">
      <h1>Registracija</h1>
    </h:panelGroup>
    <util:registracija
      email="#{uporabnik.email}"
      geslo="#{uporabnik.geslo}"
      ime="#{uporabnik.ime}"
      priimek="#{uporabnik.priimek}"
      registracijaGumbOpis="Registracija"
      registracijaGumbAkcija="#{uporabnik.registracija}" >
      <f:validateLength minimum="6" for="gesloVnosi"/>
      <f:validateRegex pattern="[\w\.-]*[a-zA-Z0-9_]@[\w\.-]*[a-zA-Z0-9]\.
                [a-zA-Z][a-zA-Z\.]*[a-zA-Z]" for="emailVnos"/>
      <f:validator validatorId="com.util.Geslo" for="gesloVnosi"/>
      <f:actionListener type="com.util.RegistracijaPosluslec"
            for="gumbRegistracija"/>
      <f:facet name="napaka">
        <h:messages layout="table" />
      </f:facet>
    </util:registracija>
  </h:panelGroup>
</h:body>
```

Kot rezultat na namiznem računalniku dobimo komponento na sliki 5.5.

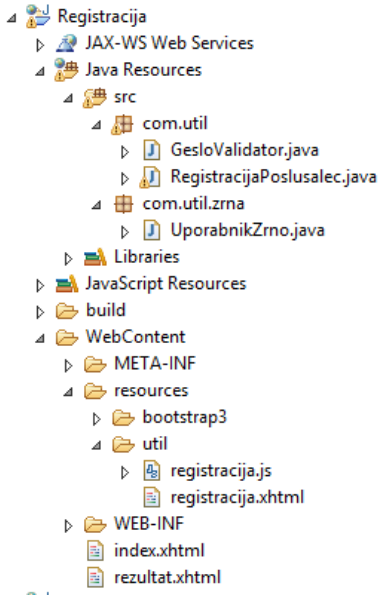
Slika 5.5: Zaslonska slika kompozitne komponente za registracijo na srednje velikem zaslonu

Z uporabo iste komponente na napravi z manjšim zaslonom, natančneje na napravi Apple Iphone v pokončnem načinu, pa dobimo komponento na sliki 5.6.



Slika 5.6: Zaslonska slika kompozitne komponente za registracijo na mobilni napravi

Struktura projekta s kompozitno komponento za registracijo je prikazana na sliki 5.7.



Slika 5.7: Struktura projekta s kompozitno komponento za registracijo

5.4 Komponenta za samodokončanje vnosa z uporabo tehnologije AJAX

Pogosto vidimo aplikacije, ki zahtevajo izbiro oz. vnos enega elementa izmed končne množice možnih izbir. Primer tega je vnos pošte, mesta, izdelovalca, tipa izdelka itd. Ko imamo manjše število možnih izbir, ponavadi uporabimo element izbirni meni (npr. meni `h:selectOneMenu` v JSF). Vendar pri velikem številu možnih izbir postane zadeva za uporabnika zelo nepregledna in posledično njegova uporabniška izkušnja slaba. Zato je v takem primeru smiselno implementirati vnosno polje, v katerega uporabnik vnese tekst, pri čemer se mu sproti prikazujejo možne izbire in mu tako ni potrebno vnesti celotne besede. Izdelava takšnih komponent pa je pri razvijalcih nepriljubljena, saj zaradi realizacije odzivnosti zahteva kar nekaj programiranja in uporabo tehnologije AJAX. Zato je smiselno razviti kompozitno komponento, ki bo enkapsulirala vse delovanje tehnologije AJAX v ozadje in jo bo uporabnik lahko le vključil v svojo stran in povezal z množico možnih izbir. Komponento bo uporabnik uporabil na način, prikazan v izseku 5.20.

Izsek kode 5.20: Uporaba komponente za samodokončanje vnosa

```
<util:samodokoncaj vrednost="#{uporabnik.izbira}"
      seznamV="#{zrno.seznamIzbir}" />
```

Implementacija same komponente je preprosta, uporabimo vnosno polje in izbirni seznam `h:selectOneListbox`, ki se bo prikazal ob tipkanju v vnosno polje in skril ob izbiri elementa iz seznama. Komponento ovijemo v odzivni odsek in izpostavimo atributa `vrednost` (izbran element) ter `seznamV` (seznam možnih izbir) (glej izsek 5.21).

Izsek kode 5.21: Datoteka `samodokoncaj.xhtml` (definicija komponente)

```
<composite:interface>
  <composite:attribute name="vrednost" required="true"/>
  <composite:attribute name="seznamV" required="true"/>
</composite:interface>
<composite:implementation>
  <h:panelGroup layout="block" class="container-fluid">
    <h:inputText id="input" value="#{cc.attrs.value}" />
    <h:selectOneListbox id="listbox" style="display: none">
      <f:selectItems value="#{cc.attrs.seznamV}"/>
    </h:selectOneListbox>
  </h:panelGroup>
</composite:implementation>
```

Da bi omogočili odzivnost, na vnosno polje registriramo dva rokovalnika JavaScript – funkcijo `posodobiSeznam()`, ki bo ob tipkanju ('onkeyup') sprožila posodabljanje seznama izbir (sprožila bo klice AJAX na strežnik) glede na vneseno zaporedje znakov in funkcijo `inputIzgubljenFokus()`, ki skrije seznam ob izgubi fokusa ('onblur') na vnosno polje. Hkrati pa moramo na vnosno polje registrirati še poslušalca, ki deluje na strežniški strani in bo dejansko posodobil sam seznam ob spremembi vrednosti vnosnega polja (funkcija `spremembaVrednosti()` v razredu `SamodokoncajPoslusalec`). Na seznam izbir pa registriramo poslušalca na strežniški strani, ki bo izbiro vpisal v vnosno polje (funkcija `elementIzbran()`). V elementu, ki predstavlja seznam izbir, uporabimo značko `f:ajax`, ki bo asinhrono posodobila vnosno polje (določeno z atibutom `render`) ob privzetem dogodku (klik). Funkcije JavaScript bodo realizirane v datoteki `samodokoncaj.js`, vključiti pa moramo tudi knjižnico Prototype, ki jo bodo funkcije JavaScript uporabljale. Dopolnjena definicija komponente je prikazana v izseku 5.22.

Izsek kode 5.22: Dopolnjena definicija komponente

```
<composite:implementation>
  <h:panelGroup layout="block" class="container-fluid">
    <h:outputScript library="js" name="prototype.js" />
    <h:outputScript library="js" name="samodokoncaj.js" />
    <h:inputText id="input" value="#{cc.attrs.value}"
       valueChangeListener="#{samodokoncajPoslusalec.spremembaVrednosti}"
       onkeyup="com.util.posodobiSeznam(this, event)"
       onblur="com.util.inputIzgubljenFokus(this)"/>
    <h:selectOneListbox id="listbox" style="display: none"
       valueChangeListener="#{samodokoncajPoslusalec.elementIzbran}"
    >
      <f:selectItems value="#{cc.attrs.seznamV}"/>
      <f:ajax render="input"/>
    </h:selectOneListbox>
  </h:panelGroup>
</composite:implementation>
```

Pri implementaciji funkcij JavaScript uporabimo JSF JavaScript API, s pomočjo katerega pošljemo strežniku klic AJAX [15, 30]. Pri realizaciji funkcij JavaScript si bomo pomagali s knjižnico Prototype, ki se v takih primerih izkaže kot uporabna, saj nam olajša pisanje kode [23]. Funkcija `posodobiSeznam()` kot argumenta pridobi vnosno polje in dogodek, ki je sprožil klic AJAX. Najprej sprožimo pošiljanje zahteve AJAX (funkcija `jsf.ajax.request()`), v kateri definiramo posodabljanje seznama. Njegov identifikator nam vrne pomožna funkcija `dobiIDSeznama()`, ki ga pridobi iz podanega identifikatorja vnosnega polja, tako da ohrani njegov začetni del do dvopičja in nato doda lokalni identifikator seznama. V zahtevi AJAX pošljemo še parametra z imenoma `x` in `y`, ki predstavljata koordinati levega zgornjega kota seznama, kjer bo izrisan (tj. levi spodnji kot vnosnega polja). V funkciji z ukazom `keyTimeout = window.setTimeout(ajaxZahteva, 300)` nastavimo pošiljanje zahteve AJAX 300 ms po zadnjem pritisku na tipko, da jih ne bi poslali preveč. Funkciji `posodobiSeznam()` in `dobiIDSeznama()` prikazuje izsek 5.23.

Izsek kode 5.23: Funkciji `posodobiSeznam()` in `dobiIDSeznama()`

```
posodobiSeznam: function(input, event) {
  var keyTimeout
  var ajaxZahteva = function() {
    jsf.ajax.request(input, event, {
       render: com.util.dobiIDSeznama(input),
```

```
      x: Element.cumulativeOffset(input)[0],
      y: Element.cumulativeOffset(input)[1] + Element.getHeight(input)
    })
   }
   window.clearTimeout(keyTimeout)
   keyTimeout = window.setTimeout(ajaxZahteva, 300)
},
dobiIDSeznama: function(input) {
  var clientId = new String(input.name)
  return clientId.substring(0, clientId.lastIndexOf(":")) + ":listbox"
}
```

Potrebna je tudi realizacija funkcije `vnosIzgubljenFokus()`, ki skrije seznam ob izgubi fokusa na vnosno polje. Kot argument funkcija dobi vnosno polje. V njej definiramo funkcijo `skrijSeznam()`, ki z uporabo objekta `Element` knjižnice Prototype skrije seznam. Tudi v tem primeru nastavimo zakasnitev, in sicer se seznam skrije po 150 ms od izbire elementa, da je uporabniku še vidna izbira, ki jo je izbral. Implementacijo funkcije `vnosIzgubljenFokus()` prikazuje izsek 5.24.

Izsek kode 5.24: Funkcija `vnosIzgubljenFokus()`

```
vnosIzgubljenFokus: function(input) {
  var skrijSeznam = function() {
    Element.hide(com.util.dobiIDSeznama(input))
  }
  izgubaFokusaTimeout = window.setTimeout(skrijSeznam, 150)
}
```

Potrebna je tudi realizacija poslušalca na strežniški strani. Zato ustvarimo razred `SamodokoncajPoslusalec`, ki bo vseboval ustrezne funkcije. Najprej implementiramo funkcijo `spremembaVrednosti()`, ki bo posodobila seznam izbir glede na vneseno zaporedje znakov. V funkciji najprej pridobimo element vnosno polje, ki je vir dogodka, ki je sprožil izvajanje funkcije. Nato preiščemo komponentno drevo in pridobimo seznam izbir (objekt `UISelectOne`). Če ga najdemo, pridobimo njegove izbire (objekt `UISelectItems`), ki so njegov prvi otrok, in njegove atribute. Nato želimo pridobiti množico možnih izbir. Pridobimo jo iz vrednosti atributa `seznamV` in atribut v primeru, da še ni bil nastavljen, nastavimo seznamu. Ko imamo množico izbir, pa pridobimo vneseno vrednost in iz množice izbir izluščimo tiste, ki se začnejo z istim zaporedjem znakov kot vnesena vrednost. Nato še nastavimo dobljeno množico izbir kot izbire seznama, s čimer ga dejansko posodobimo

in pokličemo pomožno funkcijo `nastaviSeznam()`, ki mu spremeni izgled. Če je seznam prazen (nima vrstic), ga skrije. Če pa seznam vsebuje elemente, pa pridobimo x in y koordinati kot vrednosti zahteve AJAX in seznam pozicioniramo na ustrezno mesto. Izsek 5.25 prikazuje implementacijo zgoraj omenjenih funkcij.

Izsek kode 5.25: Implementirana funkcija `spremembaVrednosti()` s pomožno funkcijo

```
public void spremembaVrednosti(ValueChangeEvent e) {
  UIInput input = (UIInput)e.getSource();
  UISelectOne seznam = (UISelectOne)input.findComponent("listbox");
  if (seznam != null) {
    UISelectItems izbire = (UISelectItems)seznam.getChildren().get(0);
    Map<String, Object> atributi = seznam.getAttributes();
    // pridobi mnozico vseh izbir
    String[] seznamIzbir = (String[])atributi.get("util.seznamV");
    if (seznamIzbir == null) {
       seznamIzbir = (String[])izbire.getValue();
       atributi.put("util.seznamV", seznamIzbir);
    }
    // in iz te mnozice vzemi samo ustrezne
    List<String> noveIzbire = new ArrayList<String>();
    String vnos = (String)input.getValue();
    for (String i : seznamIzbir) {
       if (i.toLowerCase().startsWith(vnos.toLowerCase())) {
          noveIzbire.add(i);
       }
    }
    seznam.setValue(noveIzbire.toArray());
    nastaviSeznam(noveIzbire.size(), atributi);
  }
}
private void nastaviSeznam(int vrstice, Map<String, Object> atributi) {
  if (vrstice > 0) {
    // pridobi x in y koordinate iz zahteve AJAX
    Map<String, String> parZahteve = FacesContext.getCurrentInstance()
      .getExternalContext().getRequestParameterMap();
    atributi.put("style", "display: inline; position: absolute; left: "
      + parZahteve.get("x") + "px;" + " top: " + parZahteve.get("y") + "px;");
  }
  else {
    atributi.put("style", "display: none;");
  }
}
```

Nato pa implementiramo še funckijo `elementIzbran()`, ki bo izbrano vrednost seznama nastavila vnosnemu polju. V njej najprej pridobimo seznam, ki je vir dogodka, ki je sprožil funkcijo. Nato v komponetnem drevesu poiščemo vnosno polje, predstavljeno z objektom `UIInput` in mu nastavimo vrednost, ki je bila izbrana v seznamu. Nato še pridobimo atribute seznama in ga skrijemo. Implementacija je prikazana v izseku 5.26.

Izsek kode 5.26: Funkcija `elementIzbran()`

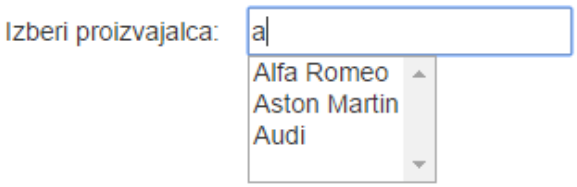
```
public void elementIzbran(ValueChangeEvent e) {
  UISelectOne seznam = (UISelectOne)e.getSource();
  UIInput input = (UIInput)seznam.findComponent("input");
  if(input != null) {
    input.setValue(seznam.getValue());
  }
  Map<String, Object> attributi = seznam.getAttributes();
  attributi.put("style", "display: none;");
}
```

Za dokončno realizacijo potrebujemo še zrno, ki bo hranilo množico možnih izbir, zrno, ki bo shranilo izbrano vrednost in stran JSF. Primer uporabe komponente na strani JSF je prikazan v izseku 5.27.

Izsek kode 5.27: Datoteka `index.xhtml` (stran JSF z uporabo komponente)

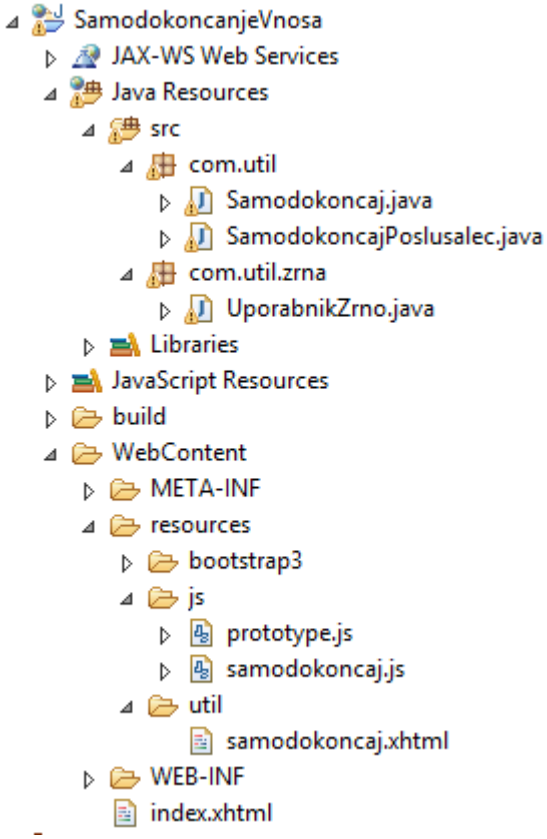
```
<h:body>
    <h:form>
      <h:panelGroup layout="block" class="container-fluid">
        Izberi proizvajalca:
        <util:samodokoncaj value="#{uporabnik.izbira}"
          seznamV="#{samodokoncaj.proizvajalci}" />
      </h:panelGroup>
    </h:form>
</h:body>
```

Kot rezultat dobimo komponento na sliki 5.8. V komponento torej vnašamo tekst in po 300 ms od zadnjega pritiska na tipko nam prikaže seznam možnih izbir, iz katerega lahko izberemo element, ki se nato vnese v vnosno polje.



Slika 5.8: Kompozitna komponenta za samodokončanje vnosa

Struktura projekta s kompozitno komponento za samodokončanje vnosa je prikazana na sliki 5.9.



Slika 5.9: Struktura projekta s kompozitno komponento za samodokončanje vnosa

Poglavje 6

Sklepne ugotovitve

Spletne aplikacije postajajo vedno bolj odzivne in dinamične, hkrati pa se pri principih razvoja le-teh kažejo težnje po čim krajšem razvoju in čim večji ponovni uporabi elementov, ki smo jih že razvili. Razvoj odzivnih komponent uspešno odgovarja na zgornje težnje in prinaša prakso modularnega razvoja aplikacij tudi v domeno spletnih aplikacij.

Na začetku dela smo predstavili javansko tehnologijo za razvoj spletnih aplikacij JSF, ki nam od verzije 2.0 naprej omogoča enostaven in hiter razvoj ponovno uporabljivih komponent. JSF deluje po principu MVC in je sestavljen iz komponent, ki se tekom življenjskega cikla aplikacije ustrezno izvedejo in preslikajo v ustrezne značke HTML. Analizirani so bili tudi drugi aspekti JSF, kot so njen življenjski cikel in dogodki, ki so pomembni pri samem razumevanju razvoja kompozitnih komponent. V naslednjem poglavju smo prikazali kompozitne komponente, njihove lastnosti in način razvoja. Ugotovili smo, da se kompozitne komponente definira v zapisu XML, kjer definiramo njihov vmesnik oz. atribute, ki jih bomo izpostavili in ki bodo omogočali njihovo prilagodljivost, in njihovo implementacijo, kjer definiramo, kako se bo komponenta preslikala v značke JSF. Nato smo predstavili tehnologije na strani odjemalca JavaScript, Bootstrap in AJAX, z uporabo katerih lahko dosežemo želeno odzivnost komponent. JavaScript je tehnologija, ki nam omogoča izvajanje kode v brskalniku odjemalca, s katero lahko manipuliramo z elementi strani ter izvajamo preverjanje vnosov oz. razne uporabne funkcije. Boostrap je ogrodje, ki vsebuje razrede CSS za izgradnjo načrta izgleda, AJAX pa skupek tehnologij, ki nam omogoča asinhrono posodabljanje

strani. V naslednjem poglavju smo razvili štiri uporabne kompozitne komponente: komponento za vnos datuma, ki uporablja pomožno zrno, komponento spinner za vnos števil, komponento za registracijo, ki se samodejno prilagaja zaslonu in komponento za samodokončanje vnosa, ki uporablja AJAX za njeno asinhrono posodabljanje.

Izkaže se, da tehnologija JSF zaradi svoje strukture in delovanja omogoča dokaj preprost in neobremenjujoč razvoj kompozitnih komponent, ki jih lahko opremimo z najrazličnejšimi funkcionalnostmi in jih lahko izpopolnimo do te mere, da so pripravljene za takojšnjo uporabo na različnih napravah, preko katerih do aplikacije dostopajo uporabniki. V diplomski nalogi smo pokazali, da je mogoče razviti zelo uporabne komponente, ki glede na razmeroma nizko težavnost razvoja uporabniku prinašajo veliko funkcionalnosti. Največje prednosti kompozitnih komponent so torej njihova ponovna uporabljivost, preprostost razvoja in praktičnost.

Slike

Tabele

Izseki izvorne kode

Literatura

[1] Advantages and disadvantages - jsf. Dostopno na: http://www.javabeat.net/advantages-and-disadvantages-jsf/. Obiskano: 5.7.2014.

[2] Bootstrap. Dostopno na: http://getbootstrap.com/. Obiskano: 15.7.2014.

[3] Bootstrap tutorial. Dostopno na: http://www.tutorialspoint.com/bootstrap/. Obiskano: 15.8.2014.

[4] Building web applications with ajax. Dostopno na: http://webdesign.about.com/od/ajax/a/aa101705.htm. Obiskano: 21.7.2014.

[5] Don't repeat yourself. Dostopno na: http://c2.com/cgi/wiki?DontRepeatYourself. Obiskano: 11.7.2014.

[6] Getting started with contexts and dependency injection and jsf 2.x. Dostopno na: https://netbeans.org/kb/docs/javaee/cdi-intro.html. Obiskano: 6.7.2014.

[7] Glassfish server. Dostopno na: https://glassfish.java.net/. Obiskano: 23.7.2014.

[8] Html5 introduction. Dostopno na: http://www.w3schools.com/html/html5_intro.asp. Obiskano: 10.8.2014.

[9] Introduction to facelets. Dostopno na: http://docs.oracle.com/javaee/7/tutorial/doc/jsf-facelets.htm. Obiskano: 11.7.2014.

[10] The java ee 7 tutorial. Dostopno na: http://docs.oracle.com/javaee/7/tutorial/doc/home.htm. Obiskano: 23.7.2014.

[11] Javascript libraries. Dostopno na: http://www.w3schools.com/js/js_libraries.asp. Obiskano: 20.7.2014.

[12] Javaserver faces. Dostopno na: http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html. Obiskano: 5.7.2014.

[13] Javaserver faces html tags reference. Dostopno na: http://www.exadel.com/tutorial/jsf/jsftags-guide.html. Obiskano: 7.7.2014.

[14] Javaserver faces(jsf) tools project - eclipse. Dostopno na: http://www.eclipse.org/webtools/jsf/main.php. Obiskano: 23.7.2014.

[15] Jsdoc reference - jsf.ajax - oracle documentation. Dostopno na: http://www.eclipse.org/webtools/jsf/main.php. Obiskano: 30.8.2014.

[16] Jsf - application events. Dostopno na: http://www.tutorialspoint.com/jsf/jsf_applicationevents_tag.htm. Obiskano: 11.7.2014.

[17] Jsf 2 fu: Best practices for composite components. Dostopno na: http://www.ibm.com/developerworks/library/j-jsf2fu0111/. Obiskano: 11.7.2014.

[18] Jsf core tag reference. Dostopno na: http://www.jsftoolbox.com/documentation/help/12-TagReference/core/f_facet.html. Obiskano: 16.7.2014.

[19] Jsf kickstart tutorial. Dostopno na: http://www.exadel.com/tutorial/jsf/jsftutorial-kickstart.html. Obiskano: 10.9.2014.

[20] The life cycle of a javaserver faces page. Dostopno na: http://docs.oracle.com/javaee/1.4/tutorial/doc/JSFIntro10.html. Obiskano: 10.7.2014.

[21] Model-view-controller architecture. Dostopno na: http://pic.dhe.ibm.com/infocenter/radhelp/v9/index.jsp?topic=%2Fcom.ibm.etools.jsf.doc%2Ftopics%2Fcmvc.html. Obiskano: 2.7.2014.

[22] Official less website. Dostopno na: http://http://lesscss.org/. Obiskano: 2.9.2014.

[23] Prototype javascript framework. Dostopno na: http://prototypejs.org/. Obiskano: 31.8.2014.

[24] Using annotations to configure managed beans. Dostopno na: `http://docs.oracle.com/javaee/6/tutorial/doc/girch.html`. Obiskano: 6.7.2014.

[25] Validation in jsf. Dostopno na: `http://incepttechnologies.blogspot.com/p/validation-in-jsf.html`. Obiskano: 15.7.2014.

[26] What are the benefits of mvc? Dostopno na: `http://blog.iandavis.com/2008/12/09/what-are-the-benefits-of-mvc/`. Obiskano: 28.6.2014.

[27] David Flanagan. *JavaScript: the definitive guide*. ≫ O'Reilly Media, Inc.≪, 2006.

[28] Jesse James Garrett et al. Ajax: A new approach to web applications. 2005.

[29] Richard Hightower and ArcMind CTO. Jsf for nonbelievers: Jsf component development. *IBM developerWorks*, 2005.

[30] James Holmes and Chris Schalk. *JavaServer faces: the complete reference*. McGraw-Hill, Inc., 2006.

[31] Cay S Horstmann and David Geary. *Core JavaServer Faces*. Prentice Hall, 2010.

[32] Anghel Leonard. *JSF 2.0 Cookbook*. Packt Publishing Ltd, 2010.